Towards Building Blocks for Agent-Oriented Programming

Standardizing Interpreters, Environments and Tools

Doctoral Thesis
(Dissertation)

to be awarded the degree of
Doctor rerum naturalium (Dr. rer. nat)

submitted by

Tristan Behrens

from Osterode am Harz

approved by

Faculty of Mathematics/Computer Science and
Mechanical Engineering
Clausthal University of Technology

Date of oral examination:	February 1st 2012
Chief reviewer:	Prof. Dr. rer. nat. habil. Jürgen Dix
Reviewer:	Prof. Dr. rer. nat. Jörg P. Müller
External reviewer:	Prof. Dr. Cees Witteveen

"Das Bedenklichste ist, dass wir noch nicht denken; immer noch nicht, obgleich der Weltzustand fortgesetzt bedenklicher wird. Dieser Vorgang scheint freilich eher zu fordern, dass der Mensch handelt und zwar ohne Verzug, statt in Konferenzen und auf Kongressen zu reden und sich im bloßen Vorstellen dessen zu bewegen, was sein sollte und wie es gemacht werden müsste. Somit fehlt es am Handeln und keineswegs am Denken."

Martin Heidegger, *Was heißt Denken?*

Abstract

The main concern of this thesis is multi-agent programming, which is preoccupied with the question *how to practically distribute intelligence?*, and thus belongs to the artificial intelligence branch of computer science. To be more precise, the focus of this thesis lies on BDI-based multi-agent programming. This means programming agents that have mental states (beliefs, desires and intentions) and capabilities (deliberation and means-end reasoning): A situation that resembles human cognition on an abstract level.

We contribute to two aspects arising in this research field. The first one is the idea of using standardization in order to support modularity and reusability. That is, identifying components that can be extracted, shared and reused. The second one is heterogeneity: Establishing multi-agent systems that are populated by agents which are implemented by means of different programming paradigms. That is, a state of affairs which is built up in order to exploit complementary functionalities and capabilities, while sharing a single environment.

The main contributions of this thesis are (standardized) interfaces for interpreters, environments and tools. Interpreters are considered to encapsulate individual agents as objects and schedule/facilitate their evolution. Environments represent an external state and define how agents connect to and access sensory and effectoric capabilities for querying and updating that state. Tools are considered to be pieces of software that are capable of querying and evaluating both the internal state of agents and of environments in order to fulfill a distinct and useful purpose. This thesis contains a formalization that reflects our approach, followed by several practical applications of the very formalization.

Zusammenfassung

Das Hauptinteresse dieser Arbeit, die sich mit der Frage *Wie verteilt man Intelligenz?* auseinander setzt, gilt der Entwicklung von Multiagentensystemen. Damit befindet man sich in der Informatik im Zweig der Künstlichen Intelligenz. Genauer gesagt liegt das Hauptaugenmerk dieser Arbeit auf der BDI-basierten Entwicklung von Multiagentensystemen. Dort werden Agenten programmiert, die sich durch einen mentalen Zustand (*beliefs, desires, intentions*) und mentale Fähigkeiten (*deliberation* und *means-end-reasoning*) auszeichnen, wobei beides den menschlichen Geistesfähigkeiten auf einer abstrakten Ebene ähnelt.

Es wird ein Beitrag zu zwei wichtigen Kernfragen des erwähnten Forschungsbereiches geleistet. Die erste betrifft die Idee der Standardisierung zur Förderung von Modularität und Wiederverwendbarkeit. Dies wird interpretiert als die Identifikation von Komponenten die extrahiert, geteilt und wiederverwendet werden können. Die zweite Kernfrage betrifft die Heterogenität, die mit Multiagentensystemen gleichzusetzen ist, in welchen eine Umgebung von verschiedenen Agenten bevölkert wird, die wiederum mit Hilfe von verschiedenen Programmierparadigmen entwickelt worden sind. Dies dient der Ausnutzung komplementärer Funktionalitäten und Fähigkeiten.

Die Hauptbeiträge sind (standardisierte) Schnittstellen für Interpreter, Umgebungen und Werkzeuge. Interpreter kapseln Agenten in Form von Objekten und planen/ermöglichen deren Evolution. Umgebungen repräsentieren einen externen Zustand und definieren, wie Agenten diesen über Sensoren und Aktuatoren wahrnehmen und manipulieren können. Werkzeuge schließlich sind Softwarekomponenten, welche in der Lage sind sowohl den internen Zustand von Agenten als auch den der Umgebung zu inspizieren und das Ergebnis einem sinnvollen Zweck zuzuführen. Diese Arbeit beinhaltet einen formalen Ansatz, der durch mehrere praktische Anwendungsbeispiele ergänzt wird.

Contents

Preface

For me, as a man who has been preoccupied for quite some time with trying to become a scientist, the last couple of years proved to be a great opportunity for my personal development and growth, as promised by my supervisor Jürgen Dix. Jürgen is a remarkable individual who, as a teacher, has a very distinct approach of leading when necessary while allowing for a maximum of liberty where possible. I have not and do not mind, because this approach payed off more than I expected. Jürgen taught me a lot of valuable lessons, which I more than happily incorporated into my everyday working life. When it comes to my research area, he allowed me to have a look around before settling for a topic, which was a very formative experience.

At this point, I also want to thank Koen Hindriks a lot. Coincidentally he has another approach to teaching PhD-students that perfectly complements Jürgen's. Koen's approach can be described as leading where possible while allowing for liberty when necessary. For me, there is absolutely no reason to complain about any of the two modi operandi and I am happy to admit that I got the most out of both.

Beyond my endeavors at the "company" I also owe a lot to my longtime companion Anja and to my parents. Anja had to endure a lot, which mostly had to do with computer science, but nevertheless helped me to become a better man. My parents, on the other hand, always supported me in my efforts, no matter how odd my plans were, including the idea to study computer science. I owe them a lot.

I also want to express my sincere gratitude towards my current closest colleagues Nils, Michael and Federico who make the hard work at the "company" an enjoyable and delightful experience.

Finally, I would like to thank the MAPC organizers, who endured as much as I did with every iteration of the contest, and the MAPC participants, who made every organizing itch and pain go away in the first second of the tournament. My colleagues at Clausthal University of Technology, the staff and students were and are great opportunities to learn more and new things, which I cherish greatly. The ProMAS community, on the other hand, has many open-minded people with great and inspiring personalities, which I cherish, too. And of course I would like to express my deeply felt gratitude towards Apple Inc. for creating more than magical products – I never wanted to believe that there are computers that just work, but now I know that such machines indeed exist. To say it with the words of the late Steve Jobs:

> "You've got to start with the customer experience and work back toward the technology – not the other way around."

1. Introduction

Computer science is the science of the algorithmic processing of information. *Artificial intelligence* is the area of computer science that deals with the study and design of intelligent agents. *Agent-oriented programming* (AOP) was coined by Yoav Shoam [57] and is a programming paradigm that promotes a social view of computing, where computation corresponds to agents that are constituted by mental components communicate, cooperate and coordinate in a shared environment. *BDI-based programming* is concerned with programming agents that are based on Michael Bratman's model of human practical reasoning [20], which defines that the mental state of an agent consists of *beliefs*, *desires* and *intentions*.

This thesis is situated in the sub-area of artificial intelligence that deals with BDI-based multi-agent systems, We consider two core issues:

- *modularity/reusability*, that is modular programming for the sake of reducing the development and testing effort, when implementing multi-agent systems and multi-agent platforms, by identifying and designing interchangeable components, called modules, and
- *heterogeneity*, that is developing and executing multi-agent systems that consist of agents implemented in different agent-oriented programming languages and executed by respective interpreters that share a single environment.

Ideally, when dealing with multi-agent systems/platforms it would be beneficial to be able to have access to a repository of *APL building blocks*. By this we mean that the used and reused components all comply to a specific standard and thus can easily be interchanged between different projects. The most important of all implied benefits of this set-up would be the reduction of development effort. On top of that, it would be easier for newcomers to create new APL/MAS projects when being able to resort to already available resources. Of course this would also imply an increase of software reliability when components are subject to a wider use.

We identify three levels of APL building blocks: 1. the agent level, 2. the interpreter level and 3. the heterogeneous MAS level.

On the agent level (see Figure 1.1), each agent is assumed to be constituted by building blocks for mental attitudes, that is the agent's model of the world and ideas what to achieve in it, and a deliberation cycle that evolves these mental attitudes over time, while interacting with other agents and an environment. Ideally, this assumption would allow developers to exchange knowledge-representation languages, that is for example, that a single building block for the Prolog-language, which can be used for belief-representation, could be used in several projects.

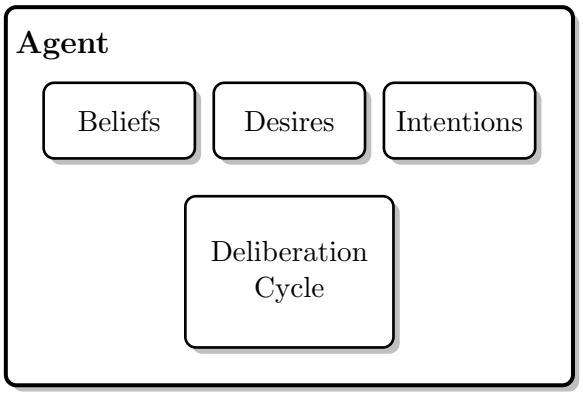


Figure 1.1.: A systematic depiction of an idealised BDI-agent. It consists of mental attitudes, that is beliefs, desires and intentions, interfaces for acting, interacting and perceiving, and of a deliberation cycle that evolves the agent's state over time.

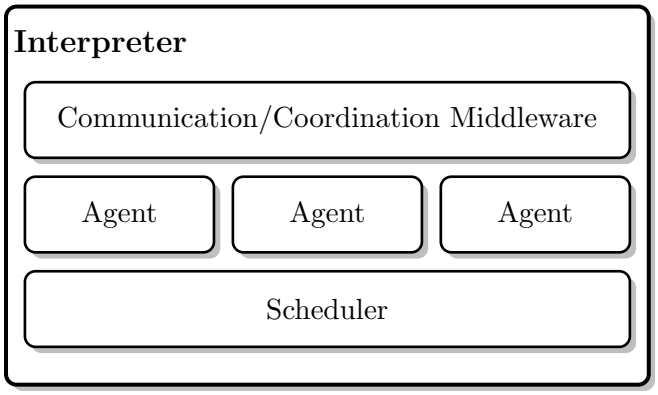


Figure 1.2.: This view shows an idealized interpreter, which contains a set of agents (three in this picture), a middleware for communication/coordination, a scheduler that schedules the execution of the agents, and an interface to an environment.

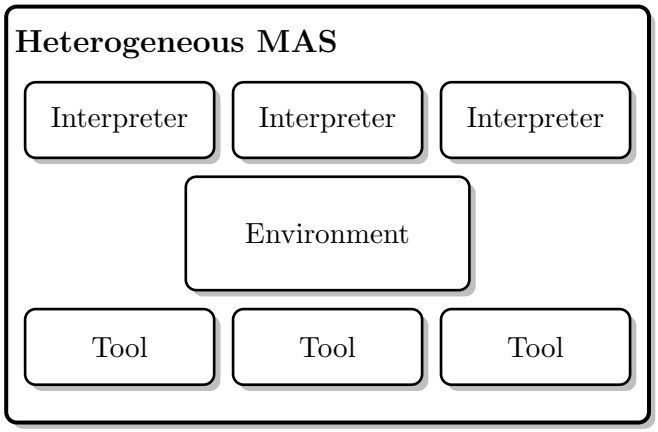


Figure 1.3.: Finally, this image depicts an idealized heterogeneous multi-agent system. The setup is constituted by a set of interpreters, a set of tools and a shared environment.

On the interpreter level (see Figure 1.2), each interpreter is assumed to consist of a set of agents, a coordination/cooperation middleware, and a scheduler. The middleware is a means for agents to coordinate their actions, which reflects the social aspect of multi-agent systems. The scheduler on the other hand coordinates the execution of the agents, which, being software components, need to be assigned time for execution, optimally using a mechanism that ensures fairness and stability.

And finally on the heterogeneous MAS level (see Figure 1.3), it is assumed that a heterogeneous multi-agent system is constituted by a set of interpreters, each hosting a set of agents, a set of tools for managing the overall execution and inspecting specific components and a shared environment. Tools like agent-inspectors that allow for the examination of the agents' mental states or state-tracers that visualize the evolution of the same are predestined to be reusable and distributable components.

With the outlined ideal of reusability/modularity integrated with heterogeneous multi-agent systems in mind, we attack three subproblems in this thesis:

1. *environment reusability/distributability* facilitated by an environment interface standard that defines a policy for agents/platforms interacting with arbitrary environments,

2. *heterogeneous multi-agent systems* facilitated by an interpreter interface that allows for plugging different interpreters into a single application and allow for a parallel execution of the same, and

3. *APL tools* facilitated by a tool interface, that permits developing distributable tools.

In this thesis we concentrate on standardizing environments, interpreters and tools. We consider interpreters as components that encapsulate individual agents as objects

and schedule/facilitate their evolution. Environments, on the other hand, are assumed to represent an external state and define how agents connect to and access sensory and effectoric capabilities for querying and updating that state. And tools are considered to be pieces of software that are capable of querying and evaluating both the internal state of agents and of environments for the sake of fulfilling a distinct and useful purpose.

Chapter 2 contains the preliminaries that constitute the basis of this thesis. Chapters 3 through 5 are the main part of this thesis. In Chapter 3 we elaborate on BDI-based agent-oriented programming and provide a comparison of three APL platforms. In Chapter 4 we formalize our approach in a stepwise manner. Chapter 5 complements Chapter 4 with a set of equivalence notions. The applications part of the thesis consists of Chapters 6 through 11. In that order, we elaborate on a standardized interface for environments, an application of an integral property of that very interface, an interface for the latests agent contest scenario that is faithful to that standard, an interface for interpreters and tools, an overview how compatibility of the interpreter and environment interfaces can be established for the three mentioned APL platforms, and a comparison case study. We conclude this thesis with Chapter 12 which provides a summary and outlines possible future work.

2. Preliminaries

As the title of this thesis suggests, we are interested both in programming multi-agent systems and in issues in the close vicinity in that area. The issues that we are interested in are *standardization*, *testing/debugging* and *comparability*. This chapter acts as an overview on that. We define the notion of *agency*, which encapsulates intelligence in objects that are called *agents*. Of course, intelligence is hardly identifiable when not applied. The intelligence of single agents becomes visible when agents interact with *environments*. The intelligence of multiple agents on the other hand becomes clear when several agents interact and cooperate while being situated in a shared environment. Of special interest for our research are agents that have an explicit mental state, which represents the agents' knowledge, goals and means to achieve the goals. This is known as the *belief-desires-intentions*-paradigm, or, in short, *BDI*.

In this chapter, we firstly motivate why and when multi-agent systems are considered to be useful. We then provide rather high-level definitions of the relevant notions. This is then followed by a brief formalization, which is heavily extended in an upcoming chapter (see Chapter 4). After that, we briefly give an idea what the BDI-paradigm is and why it is useful. We conclude this chapter with an exemplary multi-agent system and provide an overview of core issues that arise when programming multi-agent systems.

2.1. Introduction to Agents – Motivation and History

The core question with respect to the motivation for multi-agent systems is

> *"Why should we distribute intelligence?"*

According to Ferber [31], the answer to that question has multiple aspects:

- Problems are physically distributed, like for example within the field of transport engineering, that deals with the flow of vehicles/people in transport networks and the computerized analysis of the same.
- Problems are widely distributed and heterogeneous in functional terms, that is problem-solvers are available in the shape of a large number of specialists with local view and differing capabilities.
- Networks force us to take a distributed view on things, where the prime example is the internet, which is a gigantic decentralized system offering a plethora of highly specialized services.

- The complexity of problems dictates a local point of view, that is usually problems are too big to be analyzed as a whole and thus need to be divided into manageable sub-problems which are connected and interact with each other.
- Systems must be able to adapt to changes in the structure or the environment, which is implied by the high dynamics.
- Software engineering is moving towards designs using concepts of autonomous interacting units.

As we will see soon, all the requirements that follow from the aspects outlined above can be satisfied by multi-agent systems. We now consider two agent-definitions. Russel and Norvig define agents as follows [55]:

Definition 1 (Russel/Norvig's agent definition) *An* agent *is anything that can be viewed as* perceiving *its environment through* sensors *and* acting *upon that environment through* effectors.

As we can see, this definition is an adequately generic one that does not impose any assumptions about the internal structures of agents, the agent interaction, the environment and the effectors/sensors. This is appropriate for such a high level of abstraction.

A second agent definition, that assumes more, is Ferber's one [31]:

Definition 2 (Ferber's agent definition) *An agent is a physical or virtual entity that*

1. *is capable of acting in an environment,*
2. *can communicate directly with other agents,*
3. *is driven by a set of tendencies,*
4. *possesses resources of its own,*
5. *is capable of perceiving its environment to a limited extent,*
6. *has only a partial representation of its environment,*
7. *possesses skills and can offer services,*
8. *may be able to reproduce itself, and*
9. *whose behavior tends towards satisfying its objectives.*

Ferber extends the first definition by adding the notions of *proactiveness* and *social ability*, while also stating some properties of the structure of the agent itself. According to Russel/Norvig an agent is ultimately supposed to be *autonomous*, that is that its behavior is determined by its own experience, which in turn is a consequence of the agent's interaction with the world. The *behavior* of an agent is denoted by the action

that is performed after any given sequence of percepts. The job of AI-creators, on the other hand, is to design an agent program, that is a function that maps sequences of percepts to actions, while updating an internal state. Usually this is considered to be implemented in the shape and form of a *sense-think-act cycle*, which allows the agent to perpetually perceive the environment, deliberate about the state of the world and act. The agent program on the other hand is assumed to be executed on some computing device by a component that carries the name *agent architecture*. The agent itself is constituted by both the agent program and the agent architecture.

A way to describe and compare types of agents are the so called *PAGE descriptions*. The acronym stems from these agent properties:

- the sensory capabilities of the agent (**p**ercepts),
- the effectoric capabilities of the agent (**a**ctions),
- the states the agents wants to achieve (**g**oals), and
- where the agent is situated (**e**nvironment).

Usually an agent is designed to work in a class of environments, rather then to be specialized for a single purpose and/or environment. The environment is an essential component of multi-agent systems, which complements the agent components. Russel and Norvig propose that an environment is facilitated by an *environment program*, and the agent-environment interaction is rendered possible by an *environment simulator*, that 1. takes one or several agents as input and 2. arranges to repeatedly give each agent the right percepts and receive back an action [55]. The environment simulator is also responsible to keep track of the agents' *performance measures*, that is measures that evaluate the agents' performance in the environment.

While agents can be categorized by their relation to and interaction with an environment, environments can be categorized by comparing their characteristics [55]:

1. an environment is *accessible* if the agent's sensors give access to the complete state of the environment, otherwise it is called *inaccessible*,
2. an environment is called *deterministic* if its next state is determined by the current state and the actions of the agent, otherwise it is called *nondeterministic*,
3. an environment is called *episodic* if the agent's experience is divided into episodes, otherwise it is called *non-episodic*,
4. an environment is called *dynamic* if its state can change while the agent is deliberating, otherwise it is called *static*, and finally
5. an environment is called *discrete* if it has a limited number of distinct and clearly defined percepts and actions, otherwise it is called *continuous*.

A multi-agent system in general is constituted by set a of agents and a single environment. While the agents can be classified as the active components of a system – they are supposed to be proactive, reactive, and social – the environment qualifies as the passive component that facilitates the embodiment of the agents. It can be stated that [39]

- decompositions based on the agent-oriented programming paradigm are effective when it comes to partitioning the problem space of a complex system,
- in general complex systems consist of a number of related subsystems, which are usually arranged in a hierarchical manner,
- these hierarchies have several loci of control,
- the key abstractions of the agent-oriented mindset are a natural means for modeling complex systems,
- subsystems can be implemented as agents, while subsystem-interaction can be facilitated by agent-interaction, and
- the agent-oriented philosophy for modeling and managing organizational relationships is well designed for dealing with the dependencies and interactions that exist in complex systems.

2.2. Formalization of Single-Agent Systems

Continuing on our path, we work towards the definition of a *single-agent system*: a system consisting of a single agent and an environment. Most of the work presented here reflects Wooldridge's work [64], which we intend to extend in a later chapter of this thesis (see Chapter 4).

For the time being, it is assumed that at any moment in time the environment has a state. We further assume that this specific state is a member of a finite set of discrete, instantaneous environment states:

Definition 3 (environment states) *The set* $E := \{e_1, \ldots, e_m\}$ *is called the* set of environment states.

When it comes to agents, it is assumed that all of them have access to a repertoire of possible actions, that is actions that, if performed, change the state of the environment the respective agent is situated in:

Definition 4 (set of possible actions) *The set* $Ac := \{\alpha_1, \ldots, \alpha_n\}$ *is called the* set of possible actions.

Now we consider how an agent can change the state of the environment over time, that is by executing actions. Additionally, a *run* is defined as a sequence of interleaved environment states and actions:

Definition 5 (run) *A run r is a sequence*

$$r : e_0 \xrightarrow{\alpha_0} e_1 \xrightarrow{\alpha_1} e_2 \xrightarrow{\alpha_2} e_3 \xrightarrow{\alpha_3} \ldots \xrightarrow{\alpha_{u-1}} e_u$$

with $e_0, \ldots, e_u \in E$ and $\alpha_0, \ldots, \alpha_{u-1} \in Ac$. Let $\mathcal{R}$ be the set of all possible runs, then $\mathcal{R}^{Ac} \subset \mathcal{R}$ is the subset of runs that end with an action, and $\mathcal{R}^{E}$ is the subset of runs that end with an environment state.

The effect that the execution of an action has on the environment can be described by a state-transformer function, that maps runs ending with actions to subsets of environment states:

Definition 6 (state transformer function) *The function*

$$\tau : \mathcal{R}^{Ac} \rightarrow 2^{E}$$

is called state transformer function.

Note that this definition allows for formalizing nondeterministic environments, that is environments in which the outcome of an action can be uncertain. Now, with the basic definitions in place, we can define what an environment is:

Definition 7 (environment) *An* environment *is a triple*

$$Env := \langle E, e_o, \tau \rangle$$

where E is the set of environment states, $e_0 \in E$ is an initial state, and $\tau : \mathcal{R}^{Ac} \rightarrow 2^{E}$ is a state transformer function.

Agents, on the other hand, are modeled as functions that map runs to actions:

Definition 8 (agent function, set of all agents) *The function*

$$Ag : R^{E} \rightarrow Ac$$

is called agent function. *Furthermore the set $\mathcal{AG} := \{Ag, Ag', \ldots\}$ is the set of all agents.*

Single-agent systems are constituted by an agent and an environment:

Definition 9 (single-agent system) *A* single-agent system *is a tuple $SAS := \langle Ag, Env \rangle$ where Ag is an agent function and Env is an environment.*

In order to reason about the *equivalence* of individual agents it is firstly considered how two agents behave in the same environment and secondly how they behave in all environments. The main structure for reasoning about equivalence is the set of runs of an agent.

Definition 10 (set of runs of an agent, run of an agent) *The* set of runs of an agent *Ag in an environment Env is* $\mathcal{R}(Ag, Env)$*. It is assumed that this set contains only terminated runs, i.e. runs r for which* $\tau(r) = \emptyset$ *holds.*

The sequence $(e_a, \alpha_0, e_a, \alpha_1, e_2, \ldots)$ *represents a run of an agent Ag in an environment* $Env := \langle E, e_o, \tau \rangle$ *if 1.* e_0 *is the initial state of Env, 2.* $\alpha_0 = Ag(e_0)$*, and 3. for* $u > 0$ $e_u \in \tau((e_0, \alpha_0, \ldots, \alpha_{u-1}))$*, where* $Ag((e_0, \alpha_0, \ldots, e_u))$*.*

Finally, two agents are equivalent with respect to an environment if the sets of runs in the environment are the same, and two agents are equivalent if the set of runs are the same for all environments.

Definition 11 (behaviorally equivalent) *Two agents* $Ag_1, Ag_2 \in \mathcal{AG}$ *are* behaviorally equivalent with respect to an environment *Env iff* $\mathcal{R}(Ag_1, Env) = \mathcal{R}(Ag_2, Env)$*. In general the two agents are* behaviorally equivalent *if they are behaviorally equivalent with respect to all environments.*

Note, at this point, that the provided definitions are on a very straightforward level of abstraction, which is very efficient but of course leaves many questions unanswered and many issues unhandled. Thus, the model is inappropriate for formalizing the concepts of our overall goals. This circumstance, however, is not a matter of serious concern, since the formalization can be extended and adapted to our needs, which we accomplish later.

2.3. Agent Programming and BDI agents

Agent programming is a software engineering discipline that is concerned, as the name implies, with the *development of agents*. To be more precise, this is usually understood as *multi-agent systems development*, which is developing agents and an environment. An agent is considered to be a piece of software that has a couple of essential properties. The *weak notion of agency* as written down by Wooldridge and Jennings [65] is constituted by these properties:

- *autonomy* is that agents operate on their own behalf and especially without direct intervention of others, including humans,
- *proactiveness* is that agents take the initiative and show some goal-directed behavior instead of just reacting to external events,
- *reactiveness* is that agents perceive the world (the environment, other agents and itself) and react to changes in a timely fashion, and
- *social ability* is that agents interact with others.

A *stronger notion of agency* assumes the properties outlined above and, on top of that, assumes that the agents make use of *mentalistic* notions, such as beliefs, desires and intentions (BDI) [21]. The BDI-architecture is based on the idea that the mind of rational intelligent agents can be considered to be constituted by the following data-structures:

- *beliefs* that represent the agent's knowledge about the world, that is the environment and all agents, including the agent itself,
- *desires* that denote which state(s) of the world the agent currently wants to achieve, and
- *intentions* that depict what the agent actually intends to do and how he intends to do it.

Beliefs represent the *information* component of the agent's mind, while the desires stand for its *motivation* and the intentions for its *deliberation*.

While the mind of the agent consists of these mental attitudes, the agent's *control loop*, on the other hand, is some process that facilitates perceiving, the interaction of the mental attitudes and acting in the environment or interacting with other agents respectively. The BDI paradigm has its roots [64] in

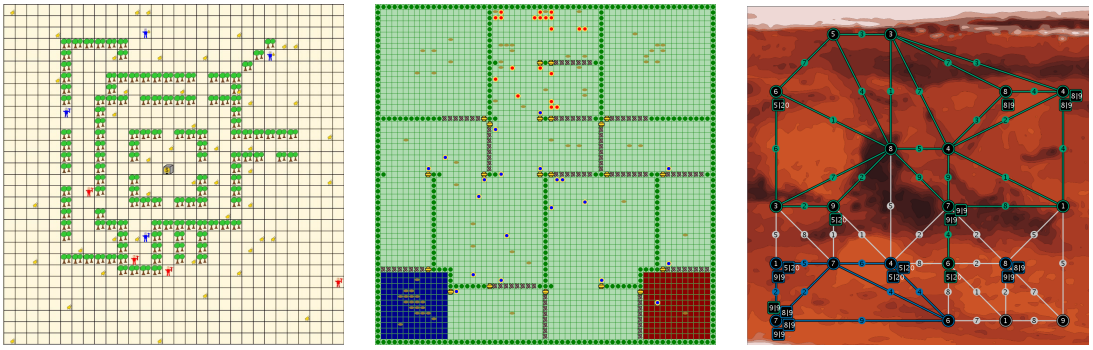
1. the field of *rational agents*, that is the field of artificial intelligence that is concerned with agents that do the right thing at the right time based on their knowledge and goals,
2. the field of *automated planning*, and usually deals with finding a course of actions, when an initial state, a set of goal states and some actions are given, and
3. the field of *decision theory*, that is deciding for the right option when several competing alternatives are given.

BDI agents are quite powerful because [21]

1. they allow for the two forms of reasoning *means-end reasoning* from automated planning and *weighing of competing alternatives* from decision theory,
2. they allow for interactions between the two, and on top of that
3. they take into account the problem of resource boundedness, whereas resource boundedness is the preliminary assumptions that agents are not capable of performing any large computation in constant time.

Because of their above mentioned capabilities, BDI agents are useful for real-time applications that fulfill these characteristics [51]:

1. *the environment is nondeterministic*, that is in each state the environment can evolve in several ways,
2. *the system is nondeterministic*, that is in each state there are potentially several different actions to perform,
3. the system can have *several different objectives at the same time*,
4. the actions/procedures that achieve the objectives best are *dependent on the state of the environment* and are *independent of the internal state of the system*,



a) b) c)

Figure 2.1.: Screenshots of the three major Contest scenarios. From left to right: the goldminers, cows and cowboys and agents on Mars.

5. the environment can only be *sensed locally*, and
6. the rate at which computations and actions can be carried out is *within reasonable bounds to the rate at which the environment evolves*.

The characteristics 1, 2, 4, 5 imply that agents need some data-structure that represent their knowledge about the world. These are the beliefs. The characteristics 3, 5 on the other hand imply that the agents need some data-structure that denotes the objects to be accomplished. These are the desires. And finally, the characteristic 6 implies that some data-structure is required that represents the currently chosen course of actions. These are the intentions. In summary, this shows that all three data-structures are necessary.

2.4. MAS Example: The Multi-Agent Programming Competition

The Multi-Agent Programming Contest (short: Contest) [24, 25, 26, 8, 9, 7] is an international competition for agent-programming. Every year, the organizers define a task that is supposed to be solved with multi-agent systems. In 2005 the first agent contest was held. Participants were asked to implement a full multi-agent system that solved a task in a given environment. The goal was to gather as many food-items on a grid-like map as possible. The participants had to implement both the agents and the environment itself.

In 2006, on the other hand, we provided the environment, which was hosted by the *MASSim* platform. *MASSim* is a platform based on a client/server architecture that contains a simulated environment and handles the agent-environment-communication which is facilitated by XML-messages transmitted via TCP/IP. On top of that, *MASSim* also handles tournament scheduling, that is scheduling matches between multiple teams.

That year the food-gathering scenario was modified and turned into the gold-miners scenario. A team of agents was supposed to explore the grid, detect gold, pick it up and drop it at a specific depot position. Agents were capable of performing a couple of actions, that is moving to an adjacent cell on the grid and pick up or drop gold. Percepts consisted of the content of the adjacent cells and their content(s). In 2007, we adapted some minor details of the scenario but did not introduce any major changes. Figure 2.1a) shows a screenshot of the scenario.

In 2008, however, the cows and cowboys scenario replaced the goldminers scenario. Again the environment was constituted by grid, but the grids were bigger. Also, the number of agents was increased while sticking to the two-teams-per-simulation idea. The rules of the environment were modified. Now the task was to gather cows, which behave according to a flocking algorithm, by frightening them into corrals. That year, there were less actions, that is agents were only capable of moving to adjacent cells. Additionally agents perceived more, namely everything which is in a visibility range. In 2009 and 2010 we only adapted the scenario a little. We increased the number of agents and improved the cow-algorithm. Figure 2.1b) shows a screenshot of the cows and cowboys scenario.

Finally, in 2011, we have established the third *MASSim* scenario, called agents on Mars. This time the topology of the environment(s) was more general, due to the use of an abstract graph instead of a graph representing a rectangular grid. On top of that the capabilities of the agents were extended by adding more actions, where in turn sets of actions were assigned to the newly introduced agent-roles. Agents, depending on their roles, were capable of moving to an adjacent node in the graph, attack and parry, probe vertices, survey edges, inspect other agents and buy equipment. Percepts now consisted of the topology of the environment in a given visibility range. Instead of performing a single task, agents were now required to reach a couple of achievements while conquering an area as big and as valuable as possible. A screenshot of the agents on Mars scenario is shown in Figure 2.1c).

Although all mentioned scenarios differ when it comes to the details of state-transitions they can be categorized in the same manner. The environments are inaccessible, that is an agent only perceives a partial state and can only act within a limited sphere of influence. Also, the environments are nondeterministic. If several agents intend to perform actions at the same time that would yield an invalid state (for example two agents standing on the same cell in the goldminers scenario) *MASSim* resolves this issue by executing a fair mechanism that is based on randomness. In general, actions can fail and percepts can be lost, which is again facilitated by another randomness-based mechanism. Additionally, the environments are non-episodic, since each simulation ends after a specified number of steps. The environments are dynamic as well. That is gold can be generated and cows move according to a flocking algorithm, which is possible without being influenced by agents. And finally, the environments are discrete, since there is a finite set of actions/percepts and a comparably huge but finite state-space.

The state evolution works as follows: At the beginning of each simulation, each agent is provided with static percepts, that contain data about the environment that does not change in the course of the simulation. In each step the server firstly provides all

agents with percepts, which contain the local view. Secondly, it gathers the actions from all agents. Thirdly, the actions are executed while maintaining consistency. Once the simulation is over, each agent perceives its outcome.

Each scenario has its own performance measure. In the goldminers scenario the number of gold pieces collected in the course of each simulation is relevant. When talking about the cows and cowboys scenario, we have to differentiate between the three incarnations. In the first one the number of collected cows during the simulation is relevant. In the second one, the number of cows in the corral in the last step of the simulation counts. And in the third one, the average number of cows in the corral is essential. In the agents on Mars scenario it is assessed how much territory each team occupies during the simulation while the amount of money earned is also taken into account.

2.5. State of the Art of AOP and Problems

Since the inception of the agent-oriented programming paradigm, a plethora of agent-oriented programming languages have been designed and many agent programming platforms have been implemented. In this section, we give a quick overview of agent-oriented platforms that are/were prominent in the Programming Multi-Agent Systems community. The following data is mainly taken from two multi-agent programming books [17, 18], which properly represent the efforts and achievements of the community spanning over more than a decade.

3APL is a BDI-based agent programming platform and has been developed to facilitate the practical development of intelligent agents. Hindriks et al. presented 3APL firstly in [36], while others defined and implemented useful extensions [28, 63]. Basic building blocks are beliefs, plans, declarative goals and reasoning rules. Rules are used to generate plans based on goals and beliefs. 3APL follows a strict separation between mental attitudes and a deliberation cycle that updates mental attitudes. 3APL had its last update in 2006 and can now be considered superseded by 2APL or at least inactive.

2APL is a continuation of the original 3APL, which is focussed on the single-agent level [23]. 2APL adds events, goals, new actions, new rules and an extension to the deliberation cycle. New actions are actions for acting in the environment, triggering the execution of new plans, and communicative action. New rules are rules for repairing failed plans and rules for reacting to external events and incoming messages. All of this requires the extension of the deliberation cycle.

The Agent Factory framework has been designed for the development and deployment of multi-agent systems [47]. The framework has been proposed by O'Hare in 1996 and has been experiencing a steady evolution since then. Its development led to the implementation of the Agent Factory Agent Programming Language (AFAPL) and its successor AFAPL2. The main building blocks for agents are goals, beliefs and commitments.

Brahms is a multi-agent modeling and simulation language, which had its original focus on modeling the work practice of humans, whereas its focus now lies on applying it as an agent programming language [60, 58]. The Brahms software makes use of a virtual machine to execute agents and provides an interface for the interaction with

humans. The main language features are mental attributes, deliberation, adaptation, social abilities, reactive and cognitive based behavior, and communication.

CLAIM [61] is a high level declarative language intended to facilitate the development of intelligent and mobile agents. It is based on the idea that a multi-agent system is constituted by a set of distributed agent-hierarchies. It combines knowledge, goals and capabilities with communication, mobility and concurrence. SyMPA [61], on the other hand, is a platform that supports the interpretation of CLAIM as a distributed language.

GOAL is a high-level programming language for rational agents, where agents base their decisions on their beliefs and goals [29]. The mental states of individual agents are constituted by knowledge, beliefs and declarative goals. The latter is a feature that distinguishes GOAL from other platforms.

IMPACT is a multi-agent framework [30]. The core concept is the idea of agentisation, that is building agents around existing legacy code. Two other features are clear semantics and easy extensibility.

IndiGolog is a programming language for embedded reasoning agents [56]. The programming language is based on the situation calculus and is supported by a high-level program execution that performs planning while agents operate. IndiGolog allows for balancing determinism and non-determinism, while also facilitating concurrency.

JACK is a commercial agent development platform that extends Java [22]. It adds new top-level declaration types for defining properties of and relationships between entities, which can be either agents, belief sets, views, events, plans and capabilities, specified by Java annotations. JACK consists of an agent-oriented programming language that is based on plans, goals and beliefs, a library for inter-agent communication and a couple of development tools.

Jade [16, 15] is a software environment that has been fully developed in Java. It consists of libraries of Java classes for agent development and a run-time environment that is required for executing multi-agent systems. Jade has been developed for the management of networked information resources. Agent communication is equivalent to exchanging asynchronous messages between heterogeneous agents and is facilitated in a FIPA-compliant [1] manner.

Jadex is a Java and XML facilitated framework for creating agents based on the BDI model [43]. Essentially it consists of a rational agent layer on top of a middleware agent infrastructure, whereas a strong emphasis lies on solid software engineering foundations. Agents are considered to be black boxes which send and receive messages, while internally agents are constituted by beliefs, goals, plans, and capabilities.

Jason [19] is a Java-based implementation of the AgentSpeak(L) [50] BDI-based agent programming language. In *Jason* agents are reactive planners, that is they react to internal or external events by instantiating and executing plans that are specified in a plan library. The developers added important multi-agent techniques and concepts to make *Jason* a practical language for developing agents.

JIAC [2] is an agent framework, whose initial area of application has been telecommunication. Its latest incarnation is JIAC V, which allows for developing agents both in Java and, more preferably, in the agent programming language JADL++, which is on a high level. Each agent consists of its own knowledge base, is executed by its own

thread and owns a set of actions that can be applied to knowledge and environment, while being able to make use of other agents' abilities.

Up to this point, we have managed to give a brief overview on the prominent agent platforms that come from the ProMAS community. This collection is inherently diverse when it comes to, amongst other things, the designated area of application, definition of what an agent is and what components it features, the underlying agent programming language, and provided development/debugging/runtime tools. We have tried to show this diversity and now arrive that the question: Given what we have outlined, which problems should be a matter of our interest? Mainly based on observation we answer:

1. *Portability of components:* All described approaches have been and are mostly still straining in different directions. Often it is the case that problems relevant for agent-oriented programming are solved on an individual basis. This includes but is not limited to providing or connecting to languages for expressing and manipulating mental attitudes, developing or making use of communication middle-wares, developing or connecting to environments, facilitating means for multi-agent profiling/debugging/testing. Such components are assessed to be portable to a certain degree and their reuse would pay-off when it comes to development time, which would be significantly reduced.

2. *Comparing AOP platforms:* On which levels of abstraction can agent-oriented programming platforms be compared and how would such a comparison be facilitated are meaningful issues. It would be interesting to find out what platforms have in common and where they differ. Interesting aspects are performance, expressibility, ease of use, extendability and portability.

2.6. Summary

In this chapter, we gave a brief but hopefully sufficient introduction to multi-agent programming. We have considered a straightforward formalization for single-agent systems, which we are going to extend later. Also, we have had a look at agent-oriented programming as a programming discipline, while focusing especially on BDI-based agent-oriented programming paradigm. Afterwards, we have elaborated on the Multi-Agent Programming Contest, which is an interesting and challenging multi-agent system. Finally, we gave an overview of the state-of-the-art of agent-oriented programming platforms. From now on we concentrate on 2APL, Goal and *Jason*.

3. BDI-Based Agent-Oriented Programming and Platform Comparison

The contents of this chapter are as follows. We give a brief overview of three exemplary BDI-based agent-oriented programming languages, namely 2APL, a practical agent programming language, GOAL, a programming language for rational agents, and *Jason*, an AgentSpeak implementation. We show examples that explain agent-programming using these languages, for example calculating Fibonacci[1]-numbers. Additionally, we briefly inspect programming on the multi-agent level and elaborate on environment programming. All of this is intended to facilitate a comparison of the three platforms, which in turn is supposed to be the preparatory work for the main part of this thesis. The outcome of the comparison will be used later to define standards for environment, tool and interpreter interfaces, based on the distilled similarities of the considered platforms, while also preparing the ground for facilitating heterogeneity. Additionally this part of the thesis is essential to reflect some of the changes that were applied to components of the platforms while preparing for this thesis.

We examine the platforms with respect to the following aspects:

- *agent programming*, that is how agents can be specified on a programming level,
- *multi-agent system programming*, that is how multi-agent systems can be defined,
- *environment programming*, that is how the platform-provided means can be employed to implement new environment,
- *agent states* including the mental state, and *agent states evolution*, that is how an agent is structured and how this structure changes during runtime, and finally
- *development tools* that support he developer in programming, running and debugging multi-agent systems.

In the course of this chapter, we consider 2APL, GOAL and *Jason*. We introduce all three platforms, concentrate on the most essential aspects, and compare them. We have chosen these three platforms to facilitate a compromise. All three are projects that are well established in the Programming Multi-Agent Systems community. We have expected and were proved right that although they are prima facie very similar, they exhibit wide dissimilarities when exposed to a close inspection. They provided a good ground to get an idea of problems that we must face, whereas results were also applicable to other platforms.

[1] Leonardo Fibonacci (c. 1170 - c. 1250)

In the context of this thesis, we cannot go into low level details when it comes to the considered agent programming languages. The appendix contains the syntax-definition of all three agent-programming languages (see Appendix A).

3.1. 2APL– A Practical Agent Programming Language

2APL [23] combines declarative and imperative style programming by integrating declarative mental attitudes, beliefs and goals, with events and plans. Beliefs represent the properties about the world that an agent beliefs to be true, goals express desirable states, and events represent information about environmental changes and agent-communication. Features of 2APL are the practical programming constructs for *generating*, *executing*, and *repairing* plans. These constructs are based on beliefs, goals and events. 2APL allows for active sensing via so called *sense actions* and passive sensing via *events* generated by the environment. It is worth mentioning that 2APL allows an agent to access multiple environments simultaneously.

Developing with 2APL means

1. implementing individual agents using the 2APL agent programming language, and
2. associating agents and environments and establishing inter-agent connectivity on the multi-agent level.

3.1.1. Agent Program Syntax and Semantics

In 2APL, each agent-program consists of eight different components. The *belief-base*, the *goal-base*, the *plan-base*, the *belief-updates*, and the *rule-base*, which consists of *planning goal rules*, *procedure call rules*, and *plan repair rules*. Figure 3.1 shows an exemplary 2APL agent program that computes Fibonacci-numbers. It consists of a belief base, a goal base and a single planning goal rule.

Beliefs belong to the declarative aspects of 2APL and encode information about the agent's world, including the environment, other agents, and the agent itself. The belief-base can be specified by means of a Prolog program, that is a set of facts and rules, where it is assumed that all the facts are ground. Note that the belief-base is started with the `Beliefs:`-statement. Here is an example of a simple fact:

```
Beliefs:
   father(abraham,isaac).
```

Of course, since we are dealing with full Prolog, the belief-base can also contain rules:

```
Beliefs:
   father(abraham,isaac).
   male(X) :- father(X,Y).
```

```
// initial belief-base
Beliefs:
  fib(1,1).
  fib(2,1).

// initial goal-base
Goals:
  calcFib(3,1000)

// belief-updates
BeliefUpdates:
  { true } Fib(N,F) { fib(N,F) }

// PG-rules
PG-rules:
  calcFib(N,Max) <- true | {
        B( fib(N-1,Z1) );
        B( fib(N-2,Z2) );
        B( is(Z,Z1+Z2) );
        Fib(N,Z);
        if B( N<Max ) then
          adopta(calcFib(N+1,Max));
        dropgoal(calcFib(N,Max))
  }
```

Figure 3.1.: A simple 2APL-agent that calculates Fibonacci-numbers. The agent knows the first two Fibonacci-numbers and knows how to calculate the others.

Goals, like beliefs, belong to the declarative side of agent programming in 2APL. They specify situations the agent wants to realize. The goal-base is a list of goal-expressions. A goal expression itself is a conjunction of ground atoms. Here is an example:

```
Goals:
   write(thesis)
```

Now we consider the definition of belief-update actions. Such actions manipulate the knowledge the agent has about its world. They can be used to store data that is either temporary, or received from other agents or environments. A belief-update is a triple, consisting of a pre-condition, an atom starting with an uppercase letter, that denotes the action itself, and a post-condition. The precondition is composed using literals, disjunction and conjunction, and specifies a set of believes that must be true in order for the action to be successful. The postcondition is a set of literals, and specifies the agent's beliefs after the execution of such an action. The `BeliefUpdates:`-statement

begins the belief-updates section of an agent program. Later we will see, how an agent can execute such actions. Here is an example of an belief-update specification:

```
BeliefUpdates:
   {true} Father(X,Y) {father(X,Y)}
```

Now we are about to consider the three types of rules, that is rules that instantiate plans. Before we do so, we show how plans can be composed, that is which basic actions can be used to create plans and how these basic actions can be concatenated.

The first basic action is the *belief-update action*, whose execution updates the belief-base by adding and removing beliefs, as specified in the respective belief-update specification. The syntax for such an action is an upper-case predicate followed by a sequence of terms. A belief-update action fails if its precondition cannot be satisfied. This action would for example add the belief `father(abraham,isaac).` to the belief-base:

```
Father(abraham,isaac)
```

The next types of actions are *belief-* and *goal-test actions*, which either check if the agent has a specific belief or if it has a specific goal. These actions block the execution of the respective plans until either the belief or the goal come true. They can also be used to instantiate variables. This following action blocks the execution of the plan until the agent believes that there is at least one son of `abraham` and then uses the first substitution for `Son` in the rest of the plan:

```
B( father(abraham,Son) )
```

This action on the other hand blocks the execution until the agent has the goal of writing something, and then substitutes the variable `D` with that something:

```
G( write(D) )
```

Also, there are four *goal-dynamics actions*, which manipulate the goal-base. Assuming that the goal-base is an ordered collection of goals, this action puts the goal at the beginning of that data-structure:

```
adopta( write(thesis) )
```

This action, on the other hand, puts the goal at the end of the goal-base:

```
adoptz( write(thesis) )
```

Goal can be dropped explicitly using three different types of basic actions. This action drops a goal from the goal-base:

```
dropgoal( write(thesis) )
```

This action drops all goals that are logical sub-goals of the given one:

```
dropsubgoals( write(thesis) )
```

And finally, this action drops all goals that have the given one as a sub-goal:

```
dropsupergoals( write(thesis) )
```

Until now, we have only discussed actions that manipulate the inner state of an agent. Now we complement them by discussing actions that change the state of the environment(s) and the state of other agents. *External actions* are capable of changing the state of the environment. They also include sensing actions. The action itself is represented via an atom. Since it is assumed that the agent cannot be certain about the outcome of an action before its execution, there is a return-value associated with each action, that can contain information about the results of the action. The external action is a construct `@env(Action,Result)`, where `env` is the name of the respective environment (there can be several ones), `Action` is the atom representing the action, and `Result` is the variable, which is substituted with the result of the action once it has been executed. Here is an example:

```
@domestic( open(door), Result )
```

Communications actions, on the other hand, are capable of changing the state of other agents. Such an action consists of a receiver, a speech act-name, an identifier representing the language of the message, an identifier representing the ontology, and the message itself. It should be noted that the language and the ontology tokens can be dropped in the default-case. Here is an example for a communication action:

```
send(agent,tell,father(abraham,isaac))
```

Now, we consider *abstract actions*. These actions are similar to procedure calls known from imperative programming. In 2APL plans can be associated with with abstract actions, the abstract actions trigger the execution of these plans. Abstract actions fail if there is no matching plan. As we will see later, a special type of rules associates abstract actions and plans. Syntactically abstract actions are atoms starting with a lowercase letter. For example:

```
goto(4,4)
```

Up to this point, we have considered the basic actions that can be used to compose plans. Now we show how these actions can actually be combined to form complex actions. Similar to established imperative programming languages, 2APL provides operators for *sequences*, *conditional choices*, and *conditional iterations*. On top of that, an operator for *non-interleaving* is available.

The sequence operator concatenates two plans and executes them one after another. Here is an example:

```
goto(4,4) ; @domestic( open(door), R )
```

The conditional choice operator takes of plans and a condition. One plan is executed only if the condition holds, the other plan is executed if this is not the case. The second (else-)case is optional. The condition is a boolean expression in respect to the agent's beliefs and goals:

```
if B( iAmAt(door) ) then {
  @domestic( open(door), R )
}
else {
  goto(4,4)
}
```

The conditional iteration operator takes a plan and a condition. Again the condition is a boolean expression in respect to beliefs and goals. The plan is executed over and over again as long as the condition holds. Here is an example of such a construct:

```
while B( closed(door) ) then {
  @domestic( open(door), R )
}
```

Finally the non-interleaving takes a arbitrary plan and executes it, ensuring that it is not interleaved with the execution of other plans. Here is an example:

```
[ goto(4,4) ; @domestic( open(door), R ) ]
```

After discussing the basic actions of plans and the operators that can be used to compose complex plans, we consider the rules that define the exact circumstances when they should be instantiated. There are three types of rules, one for *achieving goals* called *planning goal rules* (PG-rules), a second for *processing internal events and received messages* called *procedure call rules* (PC-rules), and a third for *handling and repairing failed plans* called *plan repair rules* (PR-rules).

Planning goal rules are applied when the agent has a certain set of goals and a certain set of beliefs. Syntactically such a rule consists of 1. a goal query expression, denoting a state of the goal-base, 2. a belief-query expression, denoting the state of the belief-base, and 3. a plan. Semantically the plan is instantiated if both the goal query expression (which is optional) and the belief query expression are true. When instantiating the plan, the variable substitutions from the goal and belief query expressions are applied to the plan body. We will now consider an example. The section of the agent-program that contains the planning goal rules is indicated by the `PG-rules:`-label. The following plan is instantiated when the agent has the goal of being at the position of an object and if the agent knows the exact coordinates of that position. If both requirements are fulfilled, the agent will go to that position:

```
PG-rules:
  beAt(Obj) <- pos(Obj,X,Y) | {
    goto(X,Y)
  }
```

Procedure call rules, on the other hand, are applied when the agent either executes an abstract action, like the `goto` action above, has to respond to a received message, or has to react to an external event generated by the environment. The syntactical structure is as follows. Such a rule consists of 1. an atom representing either an abstract

action, a message or an external event, 2. followed by a belief query expression, and 3. concluded with a plan. The first is an atom that is either the abstract action, or a special atom `event(...)/message(...)` denoting the event/message. The section of the agent-program that contains all procedure call rules is indicated by the `PC-rules:`-statement. Here is an example for handling a received message, that is reacting to the request by another agent to be at a specific position:

```
PC-rules:
  message(Sender,achieve,beAt(X,Y)) <- true | {
    goto(X,Y)
  }
```

Plan repair rules, finally, are applied when a plan fails to execute. Syntactically such a rule consists of 1. a first abstract plan expression, encoding the plan that has failed, 2. a belief query expression, and 3. a second abstract plan expression, that represents the repair plan. The plan repair sections in an agent-program is labelled with the string `PR-rules:`. Here is an example, that will move an agent to a position adjacent to a goal position if that cannot be reached:

```
PR-rules:
  goto(X,Y); Rest <- true | {
    goto(X,Y-1);
    Rest
  }
```

3.1.2. Deliberation Cycle

The state of an individual 2APL agent during runtime consists of its belief-base, its goal-base, its plan-base, and its event-base, while the event-base consists of the set of external events received from the environment, the set of failed plans, and the set of messages sent by other agents.

The initial state of an agent consists of its initial beliefs and goals, an initial plan-base, and the set of rules. The state of the agent changes by executing the deliberation cycle, which is constituted by the following steps:

1. **Applying all PG-rules:** Each PG-rules is applied. Although a rule could be applied several times, for example in the case of multiple relevant goals, in this step each rule is only applied once. If a rule is applied a plan is instantiated and put into the base of instantiated plans.

2. **Executing the first action of all plans:** In this step all instantiated plans are considered. Each plan's first action is executed. This scheduling-mechanism is intended to ensure fairness among the plans.

3. **Processing external events:** Now all external events, that is events generated by the environment(s) are processed. For each such event all PC-rules are considered.

Only the first rule that was found to be relevant is applied. The result of the application is that a plan is instantiated and stored in the plan-base.

4. **Processing internal events:** Internal events that correspond to the failure of plans are handled. For each failed plan the first applicable PR-rule is applied to instantiate a plan and store it in the plan base.

5. **Processing messages:** Finally all messages received from other agents are considered. Again the first applicable PC-rule is applied and an instantiated plan stored in the plan-base.

After executing these steps it is decided whether the cycle should be started all over again. The execution is stopped if no rules have been applied, no plans have been executed, and no events/messages have been processed. The execution is resumed when an external event or a message arrives.

3.1.3. MAS Syntax and Semantics

In 2APL, multi-agent systems are specified in a straightforward manner. Agent names are associated with agent programs that are loaded from provided agent program files and associated with none, one or several environments. Optionally, a fixed number of similar agents can be instantiated. We consider this example:

```
fibagent : fibagent.2apl
```

This instantiates a single agent from the file `fibagent.2apl`, while giving it the name `fibagent`. Here is a second example:

```
agent : agent.2apl 2 @someenv
```

This would instantiate two agents from the agent program file `agent.2apl`. The agent names are uniquely generated from the base name `agent`, which yields `agent` and `agent1` as agent names. The agents are then situated in the environment `someenv`.

In general, a multi-agent program specification file contains several, that is at least one, agent instantiation lines. Each line begins with an agent base-name, followed by a colon and an agent program file. The number parameter which allows multiple instantiations comes next and is optional. The optional environment component concludes and consists of an occurrence of the @ symbol followed by a comma-separated list of environments. From an implementation point of view, there is a class `APAPLBuilder` that is used to parse MAS-files, in order to load and run agents and environments. Furthermore there is a derivate of the class `apapl.Executor` that executes agents.

3.1.4. 2APL Environments

Creating environments in 2APL means to implement a class `Env` [27], that extends the class `apapl.Environment` (see below for its most relevant methods). The package name defines the environment name, which facilitates a Java Reflection loading mechanism.

Environments are distributed as jar-files. Agents can be associated with several environments, while the jar files are required to be in the user-directory.

We now most briefly consider the most relevant methods of 2APL environments:

- `addAgent(APLAgent agent)` adds an agent to the environment and stores its name and object in the hash-map. It is called by the `APAPLBuilder`. Cannot be overridden.
- `removeAgent(APLAgent agent)` removes an agent from the environment. It is called by the `APAPLBuilder`. This method cannot be overridden.
- `addAgent(String name)` should be overwritten while inheriting from the environment. It is called by the environment itself.
- `removeAgent` should be overwritten while inheriting from the environment. It is called by the environment itself.
- `throwEvent` sends an event to a set of agents. This method cannot be overridden.
- `getName` returns the name of the environment. This method cannot be overridden.
- `takeDown` is called to release the resources of the environment.
- For implementing external-actions the developer has to implement for each action a method `Term actionName(String agent, Term... params)`. These methods are called by the agent-executor.

An observation is that the environment stores agents as objects. Furthermore there is a format for exchanging data (perceive/act) between agents and the environment, based on the class `apapl.data.Term`: `APLIdent` for constants, `APLNum` for numbers, `APLFunction` for functions, and `APLList` for lists.

3.2. Goal – A Programming Language for Rational Agents

Distinguishing features of the GOAL [18] agent-programming language are declarative goals and how agents derive their actions from these goals. Mental attitudes in GOAL are goals and beliefs. These are constrained by rules from their common-sense equivalents. The *action-selection mechanism* is controlled by *action-rules* that are constructed on top of the mental attitudes. GOAL's features are:

- *declarative beliefs and declarative goals:* The agent's mental attitudes are expressed in a *knowledge representation language*, which is variable. Goals represent states of the world the agent wants to achieve.
- *blind commitment strategy:* Agents drop their goals only when they are achieved. Until then the agent commits to them.

- *rule-base action selection:* Agents select actions based on their beliefs and goals. If several actions are available, an arbitrary one is selected.
- *policy-based intention modules:* Modules structure action-rules and knowledge dedicated to achieving specific goals. This way an agent can focus on a subset of its goals using only a respective subset of its actions and knowledge.
- *communication on the knowledge-level:* Agents communicate using expressions in their knowledge representation language.

Programming in GOAL is:

- programming on the agent-level, and
- programming on the MAS-level.

3.2.1. Agent Program Syntax and Semantics

In GOAL a knowledge representation language is used to express mental-attitudes and to facilitate agent-communication. In general, GOAL is not intended to be restricted to a single knowledge representation language. In this thesis we use Prolog as an example, as the GOAL researchers did in most of their publications.

A GOAL agent-program consists of several sections. The *knowledge-base* contains static knowledge. The *belief-base*, on the other hand, contains dynamic knowledge. The *goal-base* encodes states of the world the agent wants to bring about. The *program section* consists of a set of *action rules* that facilitate the selection of actions based on the mental-attitudes. And finally the *action-specification section* contains pre- and postconditions of the available actions. Figure 3.2 depicts a simple GOAL-agent that computes the first thousand Fibonacci-numbers. The belief-base contains the first two ones. The goal-base encodes that the agent wants to compute the first thousand ones starting with the third. The first rule drops the goal of computing a Fibonacci-number once the agent begins to know it. The second computes a Fibonacci-number if it is not known and adds it to the belief-base. The third and final one makes the agent adopt the goal of computing the next number.

The knowledge-base contains the agent's static knowledge about the world, also called *conceptual* or *domain knowledge*. Syntactically, beliefs are Prolog-facts, like this one:

```
knowledge{
  father(abraham,isaac).
}
```

Since the knowledge base is based on Prolog it can also contain rules:

```
knowledge{
  father(abraham,isaac).
  male(X) :- father(X,Y).
}
```

```
main: fibonacci
{
  beliefs{
    fib(1,1). fib(2,1).
  }
  goals{
    calcFib(3,1000).
  }
  program[order=linear]{

    if goal(calcFib(N,Max)), bel(Prev is N-1), goal(calcFib(Prev,Max))
    then drop(calcFib(Prev,Max)).

    if goal(calcFib(N,Max)), not(bel(fib(N,F))), bel(Prev is N-1),
      bel(PrevPrev is Prev-1), bel(fib(Prev,FPrev)),
      bel(fib(PrevPrev,FPrevPrev)), bel(FN is FPrev + FPrevPrev)
    then insert(fib(N,FN)).

    if goal(calcFib(N,Max)), bel(fib(N,F)), bel(Next is N+1),
      not(bel(Next == Max))
    then adopt(calcFib(Next,Max)).
  }
}
```

Figure 3.2.: A simple GOAL-agent that calculates Fibonacci-numbers. The agent knows the first two Fibonacci-numbers and also how to calculate the others.

The belief-base contains dynamic knowledge, that is beliefs about the current state of the world. Syntactically the belief-base consists of a sequence of literals like the knowledge-base, but lacks rules.

The goal-base consists of a set of goals. A goal is either a literal or a conjunction of literals.

```
goals{
  write(thesis).
}
```

The program section contains a set of action-rules. In order to describe these rules we have to define what a *mental state condition* is. A mental-state condition consists of *mental atoms*. A mental atom is either an expression `bel(`*lit*`)` or `goal(`*lit*`)`. The first one is said to be true if the expression *lit*, which is a conjunction of literals including negation-as-failure, can be derived from the belief-base combined with the knowledge-base. The second one is true if *lit* can be derived from the goal-base combined with the

knowledge-base. A mental state condition is a conjunction of atoms or a negation of a mental-state condition. An action rule has the structure `if` *msc* `then` *action*. The expression *msc* is a mental-state condition. The expression *action* is either a built-in action or a user-defined one. A belief *b* can be added to the belief-base via `insert(`*b*`)` and removed via `delete(`*b*`)`. A goal *g* can be added to the goal-base via `adopt(`*g*`)` and can be removed via `drop(`*g*`)`. An agent can send a message by executing `send(`*id*`,`*lit*`)`, where *id* is an agent-identifier and *lit* is the message. User-defined actions consist of an agent-name followed by a possibly empty list of parameters. Here is an example for an action-rule:

```
program{
  if goal(beAt(Obj)), bel(pos(Obj,X,Y)) then goto(X,Y).
}
```

The action-specification section contains for each user-defined action a set of pre- and a set of postconditions. A precondition is a list of literals, and denotes the situation in which it is possible to execute the action. The postcondition expresses the effect of the action, encoded as an add/delete list. The positive literals are added to the belief base and the negative ones are removed. The negative literals are removed before the positive ones are added. Here is an example:

```
action-spec{
  goto(X,Y) {
    pre { currPos(XCurr,YCurr) }
    post { not(currPos(XCurr,YCurr)), currPos(X,Y) }
  }
}
```

In general, goals that are satisfied are deleted, but only if they have been completely achieved, which is the case if all subgoals are achieved. This reflects the blind-commitment strategy, which dictates that goals are only removed when they are achieved.

3.2.2. Deliberation Cycle

During its execution, an agent's state consists of its knowledge-base representing domain knowledge, its belief-base representing dynamic knowledge, and its goal-base representing what the agent wants to achieve, all encoded as logical atoms.

GOAL agents communicate on the knowledge level. That is they are capable of communicating in terms of beliefs and goals.

Defining the initial state of an agent is equivalent to providing 1. a declaration of a knowledge-base by means of facts and rules, 2. a set of initial beliefs in terms of a set of logical atoms, 3. a set of initial goals in terms of a set of logical atoms, 4. a set of action specifications, and 5. a program consisting of a collection of action-rules.

The deliberation cycle, which is executed indefinitely, consists of these steps:

1. **Retrieving all percepts from the environment:** A special environment method is invoked which yields all percepts that are available at the time of invocation.

2. **Determining enabled actions** In this step all actions are gathered that have an enabled precondition.

3. **Determining applicable actions** Now all actions are determined that have a mental-state condition that is satisfied.

4. **Determining options**: An option is an action that is both enabled and applicable.

5. **Selecting an option**: From the set of options select one action is selected and executed. Per default the option-selection-mechanism is randomly picking an option. This can also be customized in the agent-program.

3.2.3. MAS Syntax and Semantics

GOAL's multi-agent system specification language [49, 34] allows for both situated and non-situated agents. A multi-agent program consists of a sequence of lines, which are either environment definition or agent definition lines, while there is at most one environment definition line, which is required if agents are supposed to be situated. An agent that works without an environment can be instantiated like this:

```
fibagent:fibagent.goal
```

This instantiates an agent with the name `fibagent` from the GOAL agent program file `fibagent.goal`. A multi-agent system with a set of situated agents can be specified like this:

```
environment:somenv.jar,someenv.SomeEnv
agent1@environment:agent.goal
agent2@environment:agent.goal
agent3@environment:agent.goal
```

The first line indicates that the environment is supposed to be loaded from the jar-file `somenv.jar`, while loading the class `someenv.SomeEnv`. The subsequent agent lines are constructionally the same. Such a line begins with the agents-name, here `agent1`, followed by the @-sign and the `environment` keyword. The token after the colon represents the agent program file, which is supposed to initialize the respective agent.

3.2.4. Goal Environments

In order to use a GOAL-environment, the user has to copy a jar-file or a folder with class-files to a convenient location (e.g. the folder containing the MAS-description) and adapt the MAS-file [49].

The interface `goal.core.env.Environment` must be implemented for every class that is supposed to act as an environment. This class has to implement the methods defined therein. Agents are executed by a scheduler on the platform side that invokes the mentioned methods, which we consider now.

- `executeAction(String pAgent, String pAct)` is called by the scheduler in order to execute an action. The first parameter is the respective agent's name, the second one is a string that encodes the action. The method returns `true` if the action has been recognized by the environment, and `false` if not. An exception is thrown if the action has been recognized by the environment but its execution has failed.
- `sendPercepts(String pAgentName)` is called by the scheduler to retrieve all observations of an agent. The parameter is the respective agent's name. The method returns a list of percepts. It throws an Exception if retrieving the observations has failed.
- `availableForInput()` is called by the scheduler to determine whether the environment is ready for accepting input or not.
- `close()` is called by the platform-manager to shut down the environment.
- `reset()` is called by the platform-manager to reset the environment. It throws an exception if the reset has failed.

Note that the IDE user manual explicitly states that `executeAction` needs not to be thread-safe, i.e. the scheduler is supposed to ensure thread-safety. Thus the responsibility lies on the platform side of development.

3.3. Jason – An AgentSpeak Implementation

Jason [19] implements an interpreter for a practical extension of the AgentSpeak(L)-language [50]. *Jason* agents are *reactive planning systems*. That is, they are reactive systems that permanently run and while running react to events by executing plans that handle these events. Plans are courses of actions, the execution of plans usually leads to the execution of actions that change the state of the environment in order to achieve the agent's goals. Agent programs are effectively interpreted by the agents *reasoning cycle*, that is that the agent repeatedly perceives the environment, reasons about how to act in order to achieve its goals, and acting to change the state of the environment.

The *Jason* platform allows for implementing all the different components of a multi-agent system. Programming in *Jason* is

- programming on the multi-agent level (MAS-files),
- programming on the agent-level (agent programs),
- programming on the internal actions level (Java), and
- programming on the environment-level (Java).

Also there might be programming on the organizational level, and programming on the artifact level, which we do not examine since agent organization and environment programming are beyond the scope of this thesis.

```
// initial belief-base
fib(1,1).
fib(2,1).

// initial goal-base
!calcFib(3,1000).

// plan 1
+!calcFib(N,Max) : N < Max <-
  ?fib(N-1,Z1);
  ?fib(N-2,Z2);
  Z = Z1+Z2;
  +fib(N,Z);
  !calcFib(N+1,Max).

// plan 2
+!calcFib(N,Max) : N == Max <-
  ?fib(N-1, Z1);
  ?fib(N-2,Z2);
  Z = Z1+Z2;
  +fib(N,Z).
```

Figure 3.3.: A simple *Jason*-agent that calculates Fibonacci-numbers. The agent knows the first two Fibonacci-numbers and knows how to calculate the others.

3.3.1. Agent Program Syntax and Semantics

We now have a look at the syntax and semantics of agent programs. An agent program consists of three sections. The first is the *initial belief-base*, the second is the *initial goal-base*, and the third and final one is the *set of plans*. Figure 3.3 shows a very simple *Jason*-agent that computes a sequence of Fibonacci-numbers. The belief-base contains the first two Fibonacci-numbers. The initial goal is to calculate the first thousand numbers starting with the third one. The plan-base contains two plans. The first one calculates the next and the second one the final number.

The belief-base can be specified using three concepts that were taken from logic-programming, that is *literals*, *strong negation*, and *rules*. Beliefs represent knowledge about the world, including the environment, other agents, and the agent itself. The following expressions are examples for beliefs:

```
father(abraham,isaac).
```

Beliefs can be extended with annotations, like this:

```
father(abraham,isaac)[source(self)].
```

Annotations can be employed for a wide variety of applications. Per default the so called `source`-annotation encodes where a belief comes from. Such a belief can originate from a percept (`source(percept)`), from another agent (`source(agentname)`), or from the agent itself representing a *mental note* (`source(self)`).

Strong negation is used to make explicit that the agent beliefs a certain property not to be true:

```
~father(abraham,noah)
```

The final concept are *rules*. As usual, rules derive new facts from existing knowledge. Here is an example:

```
son(X,Y) :- father(X,Y) & male(Y).
```

In agent programming, the notion of a goal is a basic principle and usually describes properties of the states of the world the agent wants to bring about. *Jason* allows for two types of goals: *achievement goals* and *test goals*. Achievement goals can either be used *declaratively* to symbolically represent the state of affairs the agent wants to achieve, or *procedurally*, which is similar to procedures in procedural programming. This is an example for a declarative achievement goal:

```
!write(thesis).
```

A test goal on the other hand, represents an item that the agent desires to know. Here is a good example:

```
?father(X,isaac)
```

Such a goal can either be reached when it logically follows from the belief base or when a plan to achieve that knowledge succeeds.

Finally, plans generate intentions. A plan consists of two components, the *head* and the *body*. The head represents in which situation the body should be executed. The head consists of a *triggering event* and a *context*. The `tevent` token, in the structure below, is the triggering event, `context` is the context, and `body` is the plan body:

```
tevent :  context <- body .
```

A triggering event is the addition or the removal of a belief or a goal. A plan is supposed to be instantiated as a response to the appearance of an event. Since there are two types of goals and one type of beliefs, there can be six different triggering events: addition or removal of a belief, addition or removal of an achievement goal, and addition or removal of a test goal. Examples:

`+father(abraham,isaac)`	addition of a belief
`-father(abraham,isaac)`	removal of a belief
`+!write(thesis)`	addition of an achievement goal
`-!write(thesis)`	removal of an acheivement goal
`+?father(abraham,isaac)`	addition of a test goal
`-?father(abraham,isaac)`	removal of a test goal

For reactive planning systems it is common to postpone the execution of a plan until as late as possible [19]. Thus, agents can have several plans to achieve the same goal at the same time. It is the context of the plan that ensures which one of these plans will be scheduled for execution. Syntactically a context can be composed using beliefs as atomic tokens in combination with the operators for conjunction, disjunction, and default negation.

The *body* of a plan is a sequence of formulæ determining a course of actions. There are several different types of formulæ. *Acting in the environment* looks like this:

```
open(door)
```

The execution of an *achievement goal* can create a subgoal, that leads to the suspension of the original intention:

```
!go(home)
```

Alternatively a new goal can be created without the original intention to be suspended:

```
!!go(home)
```

Test goals can be created like this

```
?father(abraham,isaac)
```

Mental notes manipulate beliefs during runtime. Beliefs can be added:

```
+father(abraham,isaac)
```

They can be removed:

```
-father(abraham,isaac)
```

And beliefs can be overwritten:

```
-+father(abraham,isaac)
```

Internal actions are operators similar to system-predicates known e.g. from Prolog. They are implemented as boolean Java methods and usually stored as classes in a package:

```
libName.iActionName(X);
```

There are also standard internal actions which include but are not limited too communication actions and actions that facilitate if-then-else or loops. As usual, complex plans can be composed by concatenating formulæ using the semicolon.

3.3.2. Deliberation Cycle

In each step of its execution an agent's internal state consists of its belief-base, its goal-base, its plan-library, its set of events, its set of intentions, and its mailbox.

The belief-base consists of facts and rules. The goal-base consists of literals. The plan-library consists of plans, each plan has a triggering-event, a context and a body. Events are either perceived changes in the environment or changes in the agent's internals. Intentions are instantiated plans. And the mailbox contains messages from other agents. All these data-structures change during runtime. How they are changed is described now.

Defining the initial state of an agent means specifying the initial beliefs, goals. and plans. Initial events follow from the initial beliefs and goals. Agent programs are executed by the interpreter, which runs a *reasoning cycle*. That reasoning cycle repeatedly executes these steps in a consecutive manner:

1. **Perceiving the environment:** The agent is provided with a list of percepts, provided by the environment. It is assumed that this list consists of all percepts that are currently available.

2. **Updating the belief-base:** The agent's belief-base is synchronized with the percepts retrieved in the previous step. Percepts that are in the percepts-list but in the belief-base are added to the beliefs. Percepts that are in the belief-base but that are not in the percepts-list (any more) are removed from the beliefs. The addition and the removal of beliefs leads to the addition of events.

3. **Receiving communication from other agents:** One message is selected from the agent's mailbox. The mailbox is implemented as a message-queue, thus the head of the queue is retrieved.

4. **Selecting socially acceptable messages:** After a message has been selected, the interpreter checks whether it is socially acceptable or not. If it is not socially acceptable it will simply be deleted. Per default all messages will be accepted.

5. **Selecting an event:** The first event is selected from the set of pending ones.

6. **Retrieving all relevant plans:** A plan is relevant if its triggering event can be unified with the selected event. This step yields all plans which fulfill this requirement.

7. **Determining the applicable plans:** A plan is applicable if it is relevant and its context is implied by the belief-base. This step yields all plans that fulfill this second requirement.

8. **Selecting one applicable plan:** The first plan is selected from the set of applicable plans. The plan is instantiated and becomes an intention which is copied to the set of intentions.

9. **Selecting an intention:** a round-robin mechanism selects one intention for execution.

10. **Executing one step of an intention:** The first formula of the selected intention is executed.

Note, at this place, that the deliberation-cycle is heavily customizable. The above description represents the *default* implementation.

3.3.3. MAS Syntax and Semantics

A multi-agent program specification in *Jason* consists of the environment, a set of agents, an infrastructure and an execution control, whereas all components except for the agents are optional. In this work we leave out the infrastructure and the execution control. An exemplary specification would look like this:

```
MAS fibonacci {
  agents: fibagent;
}
```

This instantiates a single agent that is not situated in any environment. Any agent specification could also be customized by denoting amongst other things the agent-architecture, the belief-base implementation and the agent class. A special keyword is employed in order to define an environment. Another exemplary multi-agent system specification with three agents situated in an environment would look like this:

```
MAS someMAS {
  environment: SomeEnv()
  agents: agent1; agent2; agent3;
}
```

It should be noted, however, that the means for customization are not relevant for our studies and are thus not elaborated on here. This includes but is not limited to defining alternative belief base implementations and agent architectures.

3.3.4. Jason Environments

In order to create a new environment for *Jason*, a class has to be established that extends the class `jason.environment.Environment` [19]. Environments are distributed as jar-files. Each multi-agent system has at most one environment. The jar-file has to reside in the user directory. Agents are executed using infrastructures (for example centralized or Jade). Infrastructures also load agents and environments. We now consider the most important methods that constitute an environment's implementation:

- `Environment(int n)` and `Environment()` instantiate the environment with n different threads to execute actions.
- `init(String[] args)` initializes the Environment. The method is called before the multi-agent system execution. The arguments come from the multi-agent system file.

- `stop()` stops the environment.
- `isRunning()` checks whether the environment is running or not.
- `setEnvironmentInfraTier(EnvironmentInfraTier je)` sets the infrastructure tier (SACI, Jade or centralised).
- `getEnvironmentInfraTier()` gets the infrastructure tier.
- `getLoger()` [sic!] gets the logger.
- `informAgsEnvironmentChanged(Collection<String> agents)` informs the agents that the environment has changed.
- `informAgsEnvironmentChanged()` informs all agents that the environment has changed.
- `getPercepts(String agName)` returns the percepts of an agent. Includes common and individual percepts. Called by the infrastructure tier.
- `addPercept(Literal per)` adds a percept to all agents. Called by the environment.
- `removePercept(Literal per)` removes a percept from the common percepts. Called by the environment.
- `removePerceptsByUnif(Literal per)` removes all percepts from the common percepts, that match the unifier `per`.
- `clearPercepts()` clears the common percepts.
- `containsPercept(Literal per)` checks for containment.
- `addPercept(String agName, Literal per)` adds a percept to an agent. Called by the environment.
- `removePercept(String agName, Literal per)` removes a percept.
- `removePerceptsByUnif(String agName, Literal per)` removes all percepts that match the unifier `per`.
- `containsPercept(String agName, Literal per)` checks for containment.
- `clearPercepts(String agName)` clears all percepts.
- `scheduleAction(final String agName, final Structure action, final Object infraData)` schedules an action for execution.
- `executeAction(String agName, Structure act)` executes an action `act` of the agent `agName`.

An observation is that the environment allows for (external) control over action-execution strategies and provides logging-functionality (redirecting `System.out`). Note that although the environment defines these functions the two essential methods are `executeAction` and `getPercepts`, which represent a minimal agent interface.

3.4. Comparison

In this section, we compare the three introduced platforms with respect to a couple of key criteria. This is intended to yield valuable insights that guide our later designs. Here are the relevant criteria:

1. *perceiving*, that is how agents observe the state of an environment,
2. *acting*, that is how agents change the state of an environment,
3. *communication*, that is how agents change the state of other agents,
4. *deliberation*, that is how the agents decide what to do in each step of their execution,
5. *scheduling*, that is how agents are scheduled if there are several agents being executed at the same time,
6. *synchronization with the environment*, that is how the agents' execution is synchronized with the execution of the environment if there are any means to do so,
7. *environment programming*, that is how new environments can be implemented,
8. *tools*, that is which tools are available of facilitating, manipulating and monitoring the execution of the whole multi-agent system,
9. *customizability and extendability*, that is how the underlying programming language – Java – can be accessed, and
10. *heterogeneity*, that is if and how heterogeneity can be established.

The following paragraphs contain the outcome of the comparison.

Perceiving: All considered platforms have means for active sensing, that is special methods for retrieving percepts are available. GOAL and *Jason* have means for retrieving sets of all percepts currently available to agents. This is equivalent to getting all data from all sensors at once. 2APL on the other hand allows for querying selected sensors. Considering GOAL and *Jason* again, the special perceiving methods are usually triggered by the deliberation cycle, thus sensing is not implemented in the agent programs. In 2APL on the other hand the perceiving methods are usually invoked by executing actions in the agent program. *Jason* differentiates between individual percepts, that is percepts that are available to a single agent only, and global percepts, that is percepts that are

available to all agents at the same time. Additionally to the method of active-sensing outlined above, 2APL also allows for passive sensing in the sense that the environment is capable of sending percepts to the agents without them requesting that service.

Acting: Acting in 2APL/Goal/Jason is done by calling special methods. In 2APL, Goal and *Jason* acting is equivalent to executing special actions as specified in the agent program, whereas, in Goal, acting is part of the deliberation-cycle where all action-rules are evaluated and satisfied ones are executed. In Goal and *Jason* the action-to-be-performed is an object that is passed to the special method, in 2APL the method's name reflects the action's identifier. Executing an action-method in 2APL can have two outcomes. Either a return-value (an object) indicating success is returned, that might be non-trivial (e.g. list of percepts in the case of a sense-action) or terminate with an exception indicating action-failure. In Goal invoking the execute-action-method might have three outcomes. Either the return-value `true` indicating success, `false` indicating that the action has not been recognized, or an exception indicating that the action has failed. The *Jason* execute-action-method returns a `boolean`.

Communication: All three platforms provide at least two means for inter-agent communication. 2APL makes use of Jade [16] and a mechanism for local message passing between agents in several threads of a single process. Goal provides a similar local mechanism, while also featuring Jade and a third mechanism based on Java RMI [62]. Finally *Jason* allows for communication via Jade, SACI [38] and a centralized, local mechanism.

Deliberation: 2APL agents repeatedly execute five atomic steps, that is 1. creating new goals, 2. partially executing the currently instantiated plans, 3. processing external events, 4. processing internal events and 5. processing incoming messages, whereas the plan-execution involves interacting with the environment and/or other agents.

Goal agents on the other hand evolve by executing two atomic steps, that is a straightforward instance of a sense-plan-act-cycle, consisting of 1. storing percepts and incoming messages in the belief base, and 2. randomly selecting an applicable rule and executing it, which yields actions to be executed in the environment, and 3. executing the actions.

And finally, *Jason* agents execute ten atomic steps, which – because of their nature – can be summarized to five, that is 1. perceiving the environment, 2. updating the belief-base according to a belief-revision function, 3. handling incoming messages, 4. selecting an event and instantiating a plan that implements a fitting response to that plan, and 5. select and execute an instantiated plan.

Scheduling: 2APL offers two modes for executing agents. Agents can either be executed in the single-threaded mode, which runs them all, one after the other, executing the deliberation cycle one step each time. Alternatively agents can run in a multi-threaded mode, where they are executed in parallel.

Contrariwise, GOAL agents are executed by a generic scheduler that implements a round robin mechanism. Agents are dynamically selected in a fair and single-threaded manner.

Lastly *Jason* executes all agents in a multi-threaded manner. This means that for each action-to-be-executed, a thread is retained from a thread-pool, that is in this case responsible for executing the action.

Synchronization with the environment: The main point of synchronization for the considered platforms are the mechanisms for perceiving and acting. As already explained, 2APL perceives on agent program level. This means that perceptions are seized in the plan execution phase of the deliberation cycle. GOAL and *Jason*, on the other hand, perceive the environment in a phase that is distinct for agent program execution. Acting for all three platforms, however, is facilitated on the agent program execution level.

Environment programming: In all platforms, environment programming is not the main point of focus, and thus do not impose any discipline for environment programming. The GOAL interface implements no standard functionality. The abstract environment-class of 2APL only implements a mapping from agent-names to agent-objects. The abstract environment-class of *Jason* on the other hand implements more sophisticated functionality, like support for multi-threaded action-execution, dealing with the environment infrastructure tier, and/or notifiers for agents that the environment has changed.

Tools: All three platforms feature a graphical IDE that provides means for developing, executing and debugging multi-agent systems. On top of the basic developing functionalities, the platforms also provide useful tools.

2APL has an *inspector* that allows for inspecting the mental states of individual agents. A *log* is used to reflect the execution of the multi-agent system. The *state tracer* keeps track of the agents execution. And finally, the *message sender* enables the user to send messages to individual agents during runtime.

GOAL provides a *process overview* that allows for inspecting the mental states of individual agents, why also providing means that allow the developer to relay queries to the mental states at any time. The *console* is supposed to provide general messages about the execution of the multi-agent system, while also yielding information from the parser, about actions, and the output of the agents.

Jason debugging is facilitated by a *MAS console* and a *mind inspector*. The MAS console shows status and mental state updates while also yielding the output of the execution control and allowing the developer to oversee the execution of the agents. The mind inspector provides an overview for each agent in the multi-agent system, which includes, of course, the mental state. On top of that, it also allows to execute the multi-agent system for a fixed number of steps. An included agent history stores and shows the evolution of the mental states.

Customizability and extendability: *Jason* allows the developer/user to plug-in internal actions and environments packaged in jar-files. On top of that, the *Jason* platform provides a sophisticated mechanism for loading components (like e.g. a belief-base implementation) during runtime via Java reflection. Also, a mechanism for building and running entire projects is available, making the *Jason* platform a standalone IDE that allows the developer to implement agents, agent-components and environments in one and the same application without having to resort to third-party development tools.

Then again, GOAL and 2APL do not provide any mechanism for software development that go beyond (multi-)agent programming.

Heterogeneity: All three platforms make use of the Jade framework, thus establishing partial heterogeneity. This means that multi-agent systems could be set up in which agents communicate, but are not situated in one and the same environment. Currently there is no established mechanism that allows for that, although there has been a specialized attempt based on CArtAgO [52].

Interchangeability of Components: [2] Up to now, no components can be exchanged between the considered platforms.

3.5. Summary

This chapter had two main contributions. Firstly we provided a brief overview of the BDI-based agent programming platforms 2APL, GOAL and *Jason*. We concentrated on agent programming as well as on multi-agent programming. Additionally, for each platform, we were concerned with the details of agent deliberation cycles. Secondly, we provided a comparison of all three platforms. We will make use of our findings about differences and similarities in the main part of this thesis, when we will design standardized interfaces for environments, tools and interpreters.

[2] In our multi-agent programming courses we experienced first hand that it would be beneficial to have a means for interchanging environment interfaces. We provided tasks that dealt with programming agents for the agent contest using different platforms. For each platform we had a different connector, with varying functionality, performance and reliability. A single, high-quality interface would have been greatly appreciated.

4. Formalizations

In this chapter, we lay down the formalizations that are the basis for the approaches described in this thesis. Our overall goal is standardizing interpreters, environments and tools while establishing heterogeneity. Our motivation comes from the results of the previous chapter. We approach our goal in a step-wise manner, constituted by six approximations. For our formalizations, we take inspiration from Wooldridge's research (see Section 2.2 for a short summary or [64] for the complete picture), which we extend significantly.

With our first approximation, we define a rudimentary *abstract agent architecture*. In this architecture, an agent consists of a mental state and a mental state transition function. The agent's mental state evolves over time by applying that state transition function to the current mental state over and over again. Then, with our second approximation, we focus on *situatedness* and *single-agent systems* respectively. We introduce the notion of an *environment*, which consists of an environment state and an environment state transition function. An agent is situated in that environment, if the agent can perceive the environment's state to a certain extent and change it by performing actions. The situatedness of an agent is denoted by a pair of functions. A single agent and an environment constitute a *single-agent* system. With our third approximation, we concentrate on *multi-agent systems*, an extension of single-agent systems with one environment and several agents that are situated in it. After that, with our fourth approximation we work on a *conceptual separation*. We assume from real-world applications, that there is a significant gap between agents, that are percept-processors/action-generators, and objects that we identify as action-processors/percept-generators and call *entities*. With our fifth approximation, we concentrate on *heterogeneity*, which we define as having a set of agent interpreters, which execute agents, and an environment, where the agents are situated. Finally, with our sixth approximation, we focus on a *comparison* infrastructure, that allows for adding tools that are capable of inspecting agents and environments in order to fulfill a specialized purpose.

To reach a standardization of environment interfaces, interpreters and tools, we take into account what we have learned in the previous chapter. The fact that different agent programming platforms feature different notions or different levels of abstractions for key components, like knowledge representation languages, agents in general, environments et cetera requires us to come up with compromises. We think it is advisable to exploit similarities, while raising the level of abstraction when we encounter significant differences. All of this is done in order to come up with the framework we have outlined and that would establish heterogeneity.

We now begin and elaborate on the knowledge representation aspect of our approach. After that we steadily approach our final formalization.

4.1. Knowledge Representation Language

The knowledge representation language of an agent platform can be considered to be an important core component. Ferber, in his book [31], identifies five types of languages that are relevant for developing multi-agent systems:

1. *Implementation languages* are used to implement multi-agent platforms. They are commonly classical programming languages, for example Java or C/C++.

2. *Communication languages*, on the other hand, facilitate agent coordination and cooperation by providing means for exchanging data and requests.

3. *Languages for describing behaviors and the laws of the environment* specify the semantics of environments on an abstract level.

4. *Formalisation and specification languages* are employed to describe very high level notions like agent interactions or intentions.

5. The most relevant for us now are *languages for representing knowledge*, which are used to internally represent the world, that is in the mind of agents. This model is updated while the world evolves, based on the agent's inner (reflectivity) and outer awareness (perceiving). This model usually constitutes the mental state of agents, which is most interesting here. It is the key structure that facilitates the runtime state of individual agents.

In the following, we prepare a common ground for the three agent-oriented programming languages that we have in mind. The three selected platforms, as well as others which we leave unmentioned, have knowledge representation languages which are based on mathematical logic, as we have seen previously. While the considered knowledge representation languages of course differ significantly when viewed very closely, a common ground can be established. In this section we elaborate on knowledge representation languages and specify an important instance of such languages: a KR-language based on mathematical logic [4]. We firstly define the alphabet that constitutes the lowest layer of the language. Then we proceed further and define terms, that can be transformed, compared and evaluated. After that, we build logical atoms on top of terms. And finally, we compose logical formulæ using those logical atoms.

The alphabet defines different sets of symbols that we use to form more complex constructs:

Definition 12 (alphabet) *The alphabet consists of these sets of symbols*

- $\mathcal{PRED}$ *is the finite set of* predicate symbols*,*
- $\mathcal{CONST}$ *is the finite set of* constants*,*
- $\mathcal{VAR}$ *is the infinite set of* variables*, and*
- $\mathcal{FUNC}$ *is the finite set of* function symbols*.*

It should be noted that this definition, as it stands now, does not imply or require anything about or from the shape, form or structure of the four different sets. In the following, we assume a set of conventions, that we use for the sake of convenience, accessibility and readability:

1. predicate symbols are strings starting with a lower-case letter,
2. constants are either strings starting with a lower case letter, which we call *identifiers*, or floating point numbers, which we call *numerals*,
3. variables are strings starting with an upper-case letter, and
4. function symbols are either strings starting with a lower-case letter or the usual mathematical operators.

Although the sets of predicate symbols and constants are not disjoint, later on we are able to correlate arbitrary tokens using syntax. It is appropriate to provide some examples here:

Example 1 (some exemplary symbols) *These are a couple of predicate symbols:*

$$\mathcal{PRED} := \{p, q, r, fib, \ldots\}$$

These are a couple of constants:

$$\mathcal{CONST} := \{a, b, c, abraham, isaac, 1, 2, 3, 3.14159, 1.6180\ldots\}$$

These are couple of variables:

$$\mathcal{VAR} := \{X, Y, Z, A1, A2, A3, Father, Son, \ldots\}$$

And these are a couple of function symbols:

$$\mathcal{FUNC} := \{plus, minus, mult, div, mod, \ldots\}$$

On the next level we define constructs on top of the introduced sets. These are called *terms* and syntactically group different symbols in a meaningful manner:

Definition 13 (terms, ground terms) *The following expressions are terms:*

- *each constant $c \in \mathcal{CONST}$ is a term,*
- *each variable $v \in \mathcal{VAR}$ is a term,*
- *each function $f(t_1, \ldots, t_2)$, where $f \in \mathcal{FUNC}$ is a function-symbol and all t_i are terms, is a term, and*
- *each list $[t_1, \ldots, t_n]$, where all t_i are terms, is a term.*

The set of all terms is denoted by $\mathcal{TERM}$. The set of all ground terms, i.e. all terms that do not contain variables, is called $\mathcal{TERM}_{ground}$.

Here and in general, constants represent *named objects*, variables denote *ranges of objects*, functions map *tuples of objects to objects*, and lists are a *special representation of specific functions*, introduced for the sake of convenience. Again, we consider some examples:

Example 2 (some terms) *These are some exemplary terms:*

- `0`, `1.0` *and* `3.14159`,
- `red`, `green`, `blue`,
- `pos(1,2)` *and* `s(s(s(9)))`, *and*
- `[red,green,blue]` *and* `[pos(1,1),pos(2,1),pos(2,2)]` *are terms.*

Now, continuing to climb the constructional ladder, we define atoms on top of terms as follows. A proposition is constituted by a predicate symbol applied to a list of terms:

Definition 14 (logical atom) *Let $\mathcal{PRED}$ be the set of predicate symbols. Then every expression $p(t_1, ..., t_n)$ with $p \in \mathcal{PRED}$ and $t_i \in \mathcal{TERM}$ is an atom. We call each t_i a parameter. The set of all atoms is $\mathcal{ATOM}$. The set of all ground atoms, i.e. all atoms which have only ground parameters, is called $\mathcal{ATOM}_{ground}$.*

Example 3 (some logical atoms) *These are some logical atoms:*

- `father(abraham,isaac)`,
- `fib(1,1)`,
- `sum(1,1,2)`, *and*
- `difference(2,1,X)`.

It should be mentioned at this point, that processing (evaluating) a logical atom with respect to a second requires *unification*. A *unification* is an algorithm that is capable of computing *unifiers* for expressions on different levels of abstraction (as described here), whereas a unifier, if it exists, is a transformation that, if applied, makes two expressions equivalent. We do not define a unification algorithm at this point. Instead we assume that we use some standard unification [59] in the following.

In order to add another level of abstraction and to add a higher structure, we introduce compounds of logical atoms and connectives:

Definition 15 (logical formula) *Every atom $a \in \mathcal{ATOM}$ is a logical formula. Given a logical formula φ then the following expressions are logical formulæ as well:*

- $\neg\varphi$, *and*

- (φ).

Given two logical formulæ φ_1 and φ_2, then the following expressions are logical formulæ as well:

- $\varphi_1 \wedge \varphi_2$, *and*
- $\varphi_1 \vee \varphi_2$.

The set of all formulæ is called $\mathcal{FORM}$.

These following examples show a couple of logical formulæ

Example 4 (some logical formulæ) *The following expressions are logical formulæ:*

- $fib(1,1)$ *and* $\neg fib(3,1)$, *and*
- $fib(1,1) \wedge fib(2,1)$ *and* $fib(1,1) \vee fib(2,1)$.

This final token concludes the definition of a common knowledge representation language. We use this language in the rest of this chapter and in other parts of this thesis.

4.2. Mental States and Mental States Dynamics

In this section, we use the definition of the logic-based knowledge representation language, which we have outlined in the previous section, to define the mental states of agents. Our first assumption is that we use that single knowledge representation language to express all components of the mental states, which we call mental attitudes. Our second assumption is that the employed definition of mental attitudes is based on the well-known BDI model (see [20] or Chapter 3). As a reminder, the BDI model defines that an agent's mental state consists of

1. *beliefs* that express the agent's knowledge about the world,
2. *desires* that denote which state(s) of the world the agent wants to bring about, and
3. *intentions* which express means how to reach these states.

After defining what a mental state is, we show how these mental attitudes can be queried using a set of respective functions. Then we show how the mental attitudes can be updated using another set of functions. These functions are supposed to be the groundwork of an agent's deliberation process in which the agent is constantly querying and updating its mental attitudes.

We commence by providing the definition of a mental state of an agent:

Definition 16 (mental state) *A* mental state *of is a tuple*

$$ms := \langle B, D, I \rangle$$

where $B \subset \mathcal{ATOM}$ is a finite set of beliefs, $D \subset \mathcal{ATOM}$ is a finite set of desires, and where $I \subset \mathcal{ATOM}$ is a finite set of intentions. We denote with

$$\mathcal{MS} := \{ms_0, ms_1, ms_2, \ldots\}$$

the set of all mental states.

Given a mental state consisting of the three mental attitudes – beliefs, desires, intentions – it would be useful to be able to evaluate a given formula against these mental attitudes. We define that querying functionality as follows:

Definition 17 (mental state query) *Given a mental state $ms := \langle B, D, I \rangle$ and a formula $\varphi \in \mathcal{FORM}$, then we call*

- *each function*
$$q_B : 2^{\mathcal{ATOM}} \times \mathcal{FORM} \to \{\bot, \top\}$$
with $q_B : (B, \varphi) \mapsto \top$ if φ is satisfied with respect to B and $q_B : (B, \varphi) \mapsto \bot$ otherwise, a belief query function *and φ a belief query,*

- *each function*
$$q_D : 2^{\mathcal{ATOM}} \times \mathcal{FORM} \to \{\bot, \top\}$$
with $q_D : (D, \varphi) \mapsto \top$ if φ is satisfied with respect to D and $q_D : (D, \varphi) \mapsto \bot$ otherwise, a desire query function *and φ a desire query, and*

- *each function*
$$q_I : 2^{\mathcal{ATOM}} \times \mathcal{FORM} \to \{\bot, \top\}$$
with $q_I : (I, \varphi) \mapsto \top$ if φ is satisfied with respect to I and $q_I : (I, \varphi) \mapsto \bot$ otherwise, an intention query function *and φ an intention query.*

Alternatively, if the functions are fixed we write $B \models \varphi$, $B \not\models \varphi$, $D \models \varphi$, $D \not\models \varphi$, $I \models \varphi$, and $I \not\models \varphi$ respectively.

These functions evaluate sets of beliefs/desires/intentions with respect to a given formula and return if the formula is satisfied by the provided beliefs/desires/intentions. Clearly this is something that should be an integral part of an agent's deliberation mechanism, because it provides the means to investigate the agent's mental attitudes.

Again, the assumed presence of a standard unification mechanism allows us to apply the evaluation functions to formulæ, that contain variables. The existence of at least one unifier is equivalent to the formula being satisfied by a set of belief/desires/intentions. The absence, however, is equivalent to the opposite.

Example 5 *Consider a given set of beliefs consisting of a finite subset of the Fibonacci numbers*

$$B := \{fib(1,1), fib(2,1), fib(3,2), \ldots\}$$

Then the following expressions hold:

- $B \models fib(1,1)$,
- $B \not\models fib(1,2)$,
- $B \models fib(1,2) \vee fib(1,1)$, *and*
- $B \models fib(X,Y)$.

After defining how to query the mental attitudes, we specify how these can be updated:

Definition 18 (mental state update) *Given a mental state* $ms := \langle B, D, I\rangle$ *and a formula* $\varphi \in \mathcal{FORM}$*, then we call*

- *each function*
$$u_B : 2^{\mathcal{ATOM}} \times \mathcal{FORM} \to 2^{\mathcal{ATOM}}$$
with $u_B : (B, \varphi) \mapsto B'$ *a* belief update function *and* φ *a belief update,*
- *each function*
$$u_D : 2^{\mathcal{ATOM}} \times \mathcal{FORM} \to 2^{\mathcal{ATOM}}$$
with $u_D : (D, \varphi) \mapsto D'$ *a* desire update function *and* φ *a desire update, and*
- *each function*
$$u_I : 2^{\mathcal{ATOM}} \times \mathcal{FORM} \to 2^{\mathcal{ATOM}}$$
with $u_I : (I, \varphi) \mapsto I'$ *an* intention update function *and* φ *an intention update,*

Usually mental states are updated by updating the individual mental attitudes. That is, for example, that applying a belief update to a mental state yields a new mental state with the belief base updated. However, how the different update and query functions are implemented is a concern of the developers of agent architectures.

It should be noted now, that our approach is quite restricted. It is based on the underlying assumption that the mental state of an agents consists of exactly three mental attitudes, namely beliefs, desires and intentions, whereas, in general, a mental state can consist of an arbitrary number of mental attitudes. Also we assume that these mental attitudes are expressed by means of a logic-based knowledge representation language. Again, in general, it is possible to express the different mental attitudes constituting a mental state to be expressed by means of different knowledge representation languages. And finally, we do not say anything about the interplay of these mental attitudes. This interplay is a matter of theory [20] and a matter of different approaches to agent programming. We refer to [46] for an overview on the topic of mental states.

4.3. Abstract Agent Architecture

With the definition of mental attitudes in place, we are now prepared to continue our move towards defining agents platforms. As outlined earlier, we reach our goal with several approximations. Our first approximation deals with an *abstract agent architecture*. We define an agent to consist

- of a mental state that denotes what it knows about the current state of the world, that also denotes which state(s) of the world the agent would like to bring about, and how to do so, and
- a component that changes the mental state of an agent over time.

Beyond the idea of formalization, this abstract architecture can be employed to be the groundwork for a definition of *agent equivalence*, which we deal with in the next chapter.

An agent is a tuple consisting of set of possible mental states, the initial mental state and mental state transition function that maps mental states (see Definition16) to new ones. This evolution of the agent's mind is performed in a step-wise manner:

Definition 19 (agent) *An agent is a tuple*

$$Ag := \langle \mathcal{MS}, ms_0, \mu \rangle$$

where $\mathcal{MS}$ is the set of all mental states, $ms_0 \in \mathcal{MS}$ is the initial mental state, and $\mu : \mathcal{MS} \to \mathcal{MS}$ is the mental state transition function.

The mental state transition function highly depends on the considered agent, that is its agent program and the underlying agent architecture that executed the program. The mental state evolution of the agent is facilitated by repeatedly (and indefinitely) applying the mental state transition function on the current mental state, starting with the initial mental state, which is usually specified by the agent program, that is:

$$ms_0 \xrightarrow{\mu} ms_1 \xrightarrow{\mu} ms_2 \xrightarrow{\mu} \dots$$

4.4. Situatedness – Single-Agent Systems

In the previous subsection, we have defined a class of agents which function alone, that is agents that are truly independent and that do not interact with any environment and/or other agents. In practice, these agents are very rare, but still they do exist. We now facilitate our second approximation, that is establishing the interaction between a single agent and an environment.

Firstly, we provide a straightforward and simplified definition of an environment. We say that an environment is constituted by a set of states and a function that facilitates its evolution, which in turned is based on the execution of actions:

Definition 20 (environment) *Let Act be the set of available actions. An environment is a tuple*

$$Env := \langle ES, es_0, \alpha \rangle$$

where ES is a set of environment states, $es_0 \in ES$ is the initial environment state, and $\alpha : ES \times AC \to ES$ is the environment state transition function.

An environment is formally defined in manner that is comparable to the earlier definition of an agent. Note, however, that we do not assume anything about the structure of the environment states on our current level of abstraction.

Coupling this definition of an environment with that of an agent requires an extension of our previous agent definition:

Definition 21 (situated agent) *Let P be the set of available percepts. An agent is a tuple*

$$Ag := \langle \mathcal{MS}, ms_0, \mu, \pi, \sigma \rangle$$

, where $\mathcal{MS}$ is the set of all mental states, $ms_0 \in \mathcal{MS}$ is the initial mental state, $\pi : ES \to P$ is the perceive function, $\mu : MS \times P \to \mathcal{MS}$ is the mental state transition function, and $\sigma : MS \to Act$ is the action selection function.

Components that are added to the earlier definition are 1. the perceive function that maps the state of the environment to a percept, 2. a function that updates the mental state with respect to that percept, and 3. a function that maps the mental state to an action that changes the state of the environment.

A single-agent system, that is a single agent interacting with an environment is defined as such:

Definition 22 (single-agent system) *The tuple*

$$SAS := \langle Ag, Env \rangle$$

is a single-agent system.

Now, we implement a simple algorithm that facilitates the agent-environment interaction and evolves both the agent and the environment over time. We assume that es is the current state of the environment. In this case, the single-agent system evolves by repeatedly executing these steps:

1. apply the perceive function $p = \pi(es)$,
2. apply the mental state transition function $ms' = \mu(ms, p)$,
3. apply the action selection function $a = \sigma(ms')$, and
4. apply the environment state transition function $es' = \alpha(es, a)$.

This is the implementation of a simple perceive-think-act cycle. That is, the agent firstly perceives the environment, secondly it updates its mental state taking into account the percept, thirdly it selects an action taking into account the updated mental state, and finally it updates the state of the environment by executing the action.

The evolution of the single-agent system is facilitated by beginning with the initial environment state and the initial mental state, which is followed by repeatedly applying the functions, i.e.

$$(ms_0, es_0) \xrightarrow{\pi,\mu} (ms_1, es_0) \xrightarrow{\sigma,\alpha} (ms_1, es_1) \xrightarrow{\pi,\mu} (ms_2, es_1) \xrightarrow{\sigma,\alpha} (ms_2, es_2) \xrightarrow{\pi,\mu} \dots$$

Expressed in layman's terms, the execution of the single-agent system begins with an agent's mental state and an environment's state. In the first step the agent perceives and updates its mental state. In the second step the agent selects an action and updates the state of the environment. This process is then repeated ad infinitum.

4.5. Multi-Agent Systems

A slogan from the multi-agent systems community says that *there is no such thing as a single-agent system* [64]. Now, we proceed and extend our framework with the idea of a set of agents that is situated in an environment. This is our third approximation.

This approximation does not require an extension of the agent- and/or environment-definition. Instead we have to update the system's evolution mechanism. Before that, we define what a multi-agent system is:

Definition 23 (multi-agent system) *The tuple*

$$MAS := \langle \{Ag_1, \dots, Ag_n\}, Env \rangle$$

with $Ag_i := \langle MS_i, ms_{i,0}, \mu_i, \pi_i, \sigma_i, \rangle_i$ *is a multi-agent system*

The difference between the definition of a single-agent system and that of a multi-agent system is the use of a set of agents in the second instead of a single agent in the first. We now have to specify how the multi-agent system evolves. Evolving a single-agent system means perceiving, thinking and acting with respect to a single agent. This, of course, has to be changed to perceiving, thinking and acting of several agents. The multi-agent system evolves by repeatedly executing these steps:

1. select an agent Ag_i
2. apply the perceive function $p = \pi_i(es)$,
3. apply the mental state transition function $ms_i' = \mu_i(ms_i, p)$,
4. apply the action selection function $a = \sigma(ms_i')$, and
5. apply the environment state transition function $es' = \alpha(es, a)$.

The difference to the single-agent system's evolution lies in the action selection function that is applied in the first step. For the time being, this function is considered to be arbitrary and a matter of decision when moving to the practical implementation level. Usually this is a scheduling problem that needs to be solved, that is deciding which agent is allowed to execute a single iteration of its deliberation cycle and when this is the case.

Another important component of successful multi-agent systems has been left untouched until now: Inter-agent communication or messaging for short. Messaging is essential for the coordination of several agents on a multi-agent level. This usually requires the existence of and interfacing with a communication infrastructure that allows the agents to communicate (on a knowledge level or even beyond) with other agents in order to coordinate their actions.

We, on the other hand, do not intend to deal with communication infrastructures now, because such components are beyond the scope of this thesis. Messaging, however, is not impossible with the introduced framework. Messaging can always be facilitated using the environment as a medium for cooperation and coordination. That is, the agents can be equipped with special actions that pass messages to the environment, which in turn relays them to the designated recipient(s), which then receive the messages as percepts. This notion is the integral idea behind the agents and artifacts paradigm [54], in which environments are structured in a manner that yields the identification of artifacts, that work as means for agent coordination.

We now conclude this section with an exemplary run. We assume that the multi-agent system consists of two agents Ag_1 and Ag_2, with individual mental states, and an environment with a state itself. Furthermore we assume that the agent selection mechanism is facilitated by an appropriate scheduling mechanism. Now, for the sake of providing an example, we assume that this is a round robin scheduler that selects the agents one after the other. An exemplary run would be this:

$$(ms_{1,0}, ms_{2,0}, es_0) \xrightarrow{\pi_1,\mu_1} (ms_{1,1}, ms_{2,0}, es_0) \xrightarrow{\sigma_1,\alpha_1} (ms_{1,1}, ms_{2,0}, es_1) \xrightarrow{\pi_2,\mu_2}$$
$$(ms_{1,1}, ms_{2,1}, es_1) \xrightarrow{\sigma_2,\alpha_2} (ms_{1,1}, ms_{2,1}, es_2) \xrightarrow{\pi_1,\mu_1} \ldots$$

Again, we express the evolution of the multi-agent systems evolution in layman's terms. The overall execution begins with two agents' mental states and the state of the environment. The first agent is selected. In the first step the agent perceives and updates its mental state. In the second step the agent selects an action and updates the state of the environment. Then the second agent is selected allowing it to perceive, think and act. This process is then repeated.

4.6. Conceptual Separation

While solving our problem on the practical implementation level, we have encountered conceptual gap between agents and environments. When implementing, agents and environments are usually separated and distinguished objects. In general, programmatically, an agent-object cannot or should not be a part or a member of the environment-object.

This is caused by the fact that no fixed assumptions about the nature of the connection to the environment can be made. It cannot be assumed that

1. agents and environments are objects that are directly connected,
2. agents and environments run in the same process, that is the connection could be for example a TCP/IP connection, and
3. agents and environments were programmed using one and the same underlying implementation language.

We come to the conclusion that an environment interface is required, that fills the conceptual gap between agents and environments with structures that we call entities. These entities are supposed to facilitate the agents' connections to environments, which is called situatedness. This is the idea behind our fourth approximation. We define controllable entities to be the components of an environment interface that provide agents with percepts and relay actions to the environment. This is the core idea behind our fourth approximation.

We commence by extending the environment definition that we have used so far. We add a structure in the shape and form of controllable entities, which are percept generators and action processors, as opposed to agents, which we consider to be percept processors and action generators. The following definition adds controllable entities:

Definition 24 (environment with controllable entities) *Let Act be the set of available actions. $Env := \langle ES, es_0, CE \rangle$ is the environment where ES is a set of environment states, $es_0 \in ES$ is the initial environment state, and $CE := \{ce_1, \ldots, ce_n\}$ is the set of controllable entities, where each $ce_i := \langle \pi_i, \alpha_i \rangle$ is a controllable entity, whereas $\pi_i : ES \to P$ is a perceive function and $\alpha_i : ES \times AC \to ES$ is an environment state transition function.*

The introduced structural refinement or extension is the wrapping of a perceiving function and a environment state transition (acting) function to form a controllable entity. In the following, we intend to assume an environment populated by such entities, while the agents are separated and connect via an appropriate interface. The next logical step is the extension of our agent definition, for preparing it to be connected to an environment via one or several entities:

Definition 25 (disembodied agent) *Let P be the set of available percepts. A disembodied agent is a tuple*

$$Ag := \langle MS, ms_0, \mu, \sigma, ce \rangle$$

where MS is the set of all mental states, ms_0 is the initial mental state, $\mu : MS \times P \to MS$ is the mental state transition function, $\sigma : MS \to Act$ is the action selection function, and ACE is the agent's set of associated controllable entities.

The most essential change from our recent agent definition is the equipment of the agent with a set of associated controllable entities. These entities are the means for the agent to interact with the environment, as we make clear with the next definition:

Definition 26 (multi-agent system with conceptual separation)
The tuple

$$MAS := \langle \{Ag_1, \ldots, Ag_n\}, Env \rangle$$

with the set of disembodied agents $Ag_i := \langle MS_i, ms_{i,0}, \mu_i, \sigma_i, ACE_i \rangle$*, and with the tuple* $Env := \langle ES, es_0, CE \rangle$ *denoting the environment with controllable entities, is a multi-agent system with conceptual separation.*

In this definition, the connection between agents and environments is made explicit. Each agent has a set of associated controllable entities, the environment has a set of controllable entities, and the intersection between the two sets is the interface that establishes the agents' situatedness possible, which we call the *agents entities relation*.

Now, with two definitions in place that conceptually separate agents and environments when it comes to acting/perceiving, while facilitating acting/perceiving through controllable entities, it is required to update the process that evolves the agents and the environment, as follows:

1. select an agent $Ag_i = \langle MS_i, ms_{i,0}, \mu_i, \sigma_i, ACE_i \rangle$,
2. select a controllable entity $ce_j := \langle \pi_j, \alpha_j \rangle \in ACE_i$
3. apply the perceive function $p = \pi_j(es)$,
4. apply the mental state transition function $ms_i' = \mu_i(ms_i, p)$,
5. apply the action selection function $a = \sigma(ms_i')$, and
6. apply the environment state transition function $es' = \alpha_j(es, a)$.

The most important extension is the second step. After selecting an agent for execution in the same manner as before, one of the agent's controllable entities is selected. This selected entity is then used to perceive the environment by the agent accessing its sensory capabilities. Finally, the mental state is updated as usual. After that the agent selects an action, which is then executed by the agent accessing the entity's effectoric capabilities. These steps are then repeated as usual.

4.7. Heterogeneity

At this point, we have multi-agent systems in place, which are constituted by a set of agents and a single environment, whereas the interaction between the agents and the environment is facilitated by associating agents and controllable entities. Now, in our fifth approximation, we move towards *heterogeneity*.

In general, two levels of heterogeneity can be distinguished:

1. *agent* heterogeneity, that is a multi-agent system is populated by agents that are different according to a difference measure, which is for example the case if they are constituted by different agent programs, and

2. *platform heterogeneity*, that is a multi-agent system is populated by agents that are implemented and executed by different platforms.

While the first level of heterogeneity is already respected by our framework – agents can be defined with different perceive, mental state update and act functions – we focus on the second level now, in our fifth approximation. We concentrate on extracting interpreters out of platforms. We consider an interpreter to be a distinct component of an agent platform, a structure that encapsulates agents and executes them. Usually an interpreter also facilitates agent scheduling, that is assuming the presence of a set of agents, deciding how these are executed.

As our first step we define what an agent interpreter is:

Definition 27 (agent interpreter) *The tuple*

$$I := \langle \{Ag_1, \ldots, Ag_n\}, \sigma \rangle$$

where each Ag_i is an agent, and $\sigma : \mathbb{N} \to \mathcal{AG}$ is a scheduling function, is an interpreter.

Thus, an agent interpreter consists of a collection of agents that it executes and a scheduling function that selects an agent-to-be-executed at any given point of time.

The next step would be to extend our latest multi-agent definition in order to take the new interpreter notion into account:

Definition 28 (scenario multi-agent system) *The tuple*

$$MAS := \langle \{I_1, \ldots, I_n\}, Env \rangle$$

where $I_i := \langle \{Ag_1, \ldots, Ag_n\}, \sigma \rangle$ is an interpreter, each agent set is a set of disembodied agents, and Env is an environment with controllable entities is a scenario multi-agent system.

The main difference to our previous definition is the idea that sets of agents are wrapped by independent interpreters that manage the execution of their agents. We are now obliged to update the execution mechanism on top of this idea:

1. select an interpreter I_j,
2. select an agent $Ag_i = \langle MS_i, ms_{i,0}, \mu_i, \sigma_i, ACE_i \rangle$ using the interpreter's scheduling function σ_j,
3. select a controllable entity $ce_j := \langle \pi_j, \alpha_j \rangle \in ACE_i$
4. apply the perceive function $p = \pi_j(es)$,
5. apply the mental state transition function $ms_i' = \mu_i(ms_i, p)$,
6. apply the action selection function $a = \sigma(ms_i')$, and
7. apply the environment state transition function $es' = \alpha_j(es, a)$.

The only addition is the first step in which the execution mechanism selects an interpreter from the set of interpreters involved. In summary the process is to select an interpreter, select an agent, select an controllable entity, and execute one iteration of the agent's sense-think-act cycle.

4.8. Thorough Infrastructure

At this point, we have achieved platform heterogeneity by defining a notion of multi-agent systems that contains interpreters, which in turn contain agents, while still being faithful to the idea of conceptual separation, which separates agents and controllable entities.

Now, in our sixth and final approximation, we add tools to the scenario. Our goal is to establish an infrastructure that allows for a later practical definition of (standardized) interfaces for interpreters, tools and environments. Tools, which are our current consideration, are supposed to be software instruments that are somehow synchronized with the execution of whole multi-agent systems, while being equipped with the necessary capabilities to inspect the internals of agents, interpreters and environments in order to evaluate those to specific ends. In the following, we provide a definition of such a thorough infrastructure and define a proper execution mechanism.

The following definition of what a tool should be, is on a very high-level. This is the case because we do not want to assume anything about the structure beyond the ability to query interpreters, agents and environments and present the results of evaluations to the user:

Definition 29 (tool) *Any object T, that is capable of querying arbitrary interpreters, agents and environments, that is capable of processing the query-results and that is capable of presenting the outcomes to the user, is called a tool.*

This definition is very general but appropriate. Now, we merge this tool notion with the definition of multi-agent systems:

Definition 30 (scenario multi-agent system with tools) *The tuple*

$$MAS := \langle \{I_1, \ldots, I_n\}, Env, \{T_1, \ldots, T_m\} \rangle$$

where each $I_i := \langle \{Ag_1, \ldots, Ag_n\}, \sigma \rangle$ is an interpreter, each agent set is a set of disembodied agents, Env is an environment with controllable entities, and each T_j is a tool, is a scenario multi-agent system with tools.

It is assumed that in such a multi-agent system neither the agents, the interpreters nor the environment are affected by the execution of the tools. It is important to assert that the tools' capabilities are not beyond inspecting and evaluating.

With a set of tools in place, we have provided a final update to our execution mechanism:

1. select an interpreter I_j,

2. select an agent $Ag_i = \langle MS_i, ms_{i,0}, \mu_i, \sigma_i, ACE_i \rangle$ using the interpreters scheduling function σ_j,
3. select a controllable entity $ce_j := \langle \pi_j, \alpha_j \rangle \in ACE_i$
4. apply the perceive function $p = \pi_j(es)$,
5. apply the mental state transition function $ms_i' = \mu_i(ms_i, p)$,
6. apply the action selection function $a = \sigma(ms_i')$,
7. apply the environment state transition function $es' = \alpha_j(es, a)$, and
8. let all tools T_k query all agents, interpreters and environments, and present the result to the user.

The final step is the essential addition, which allocates some time for the tools to fulfill intended purposes, while the other components are at rest

4.9. Summary

This chapter contains an attempt to formalize the core of this thesis. We have identified four integral components of multi-agent systems. That means 1. agents that are capable of perceiving, thinking and acting, 2. interpreters that encapsulate several agents, 3. environments that provide percepts and perform actions via controllable entities, and 4. tools that allow the evaluation of the execution of the overall system. Thus, we have identified tree potential points of contact for the definition of standards: 1. interpreters, 2. environments, and 3. tools. We consider these throughout the thesis.

5. Equivalence Notions

In the previous chapter, we have provided an extended formalization for the multi-agent systems that we consider in this thesis. As a short reminder, we began with the formalization of individual agents based on mental states, performed some intermediate steps afterwards, and progressed towards a formalization of heterogeneous multi-agent systems with tools support. This very formalization consists of one or several interpreters, each executing one or several agents, and an environment with controllable entities, whereas one agent is associated with one or several entities. On top of that, we have outlined an execution mechanism that selects agents and executes them.

Now, we complement this effort with some work on agent equivalence. We are interested in considering different notions of agent equivalence, which are established by assuming different perspectives. We have a look at traces of 2APL/Goal/*Jason* agents and then come up with a generic version of that idea. It is assumed that any agent – we are not restricting ourselves to the three platforms – generates an extensive trace, that includes multiple and diverse basic actions. We filter down those traces and reduce tokens in order to establish different notions of equivalence.

5.1. About Traces and Agent Equivalence

When a multi-agent system is executed, each agent generates its very own individual trace, that reflects the changes of its internal state and its interaction with the world, which includes most prominently the environment and additionally other agents if present. This execution trace could include, but is not limited to

- *received percepts*, that is all information provided by or gathered from the external environment by means of sensors,
- *queries and updates* of the mental attitudes, which include inspecting the agent's mind and changing it to internal and external events,
- *(partially) executed plans*, which means scheduling plans for execution and executing them either as a whole or in a stepwise manner,
- *communication actions*, that is either receiving messages from other agents or sending messages to others,
- *access to legacy code*, which is accessing and executing pre-existing code in order to facilitate agentization and improve the performance of the agent executing the code, and

- *executing environment actions*, that is changing the state of the environment in the agent's local frame.

This list is rather general and can be considered incomplete, but gives a good impression about what a trace could be expected to consist of. Let us reconsider our general execution mechanism. It roughly consists of these six steps:

1. select an interpreter,
2. select an agent,
3. select a controllable entity that is associated with the agent,
4. employ the controllable entity to get all percepts that are available in the current state of the world,
5. update the agents mental state taking into account those percepts, which by means of abstraction could include messages relayed by the environment, and of course the agent's reflection, and
6. select and execute an action in order to update the environment.

We are most interested in the steps 4, 5 and 6, because they reflect an individual agent's dynamics and as such directly contribute to the generation of an agent's trace. Such a trace would then consist of sequences of percepts, mental state dynamics actions and external actions. We can safely assume that it is clear what percepts and actions are. More interesting are the mental state dynamics actions. We intend to be on a most general level here and only assume the following definition:

Definition 31 (mental state dynamics actions) *Every action that is the direct cause of a change in an agent's mental state is called a* mental state dynamics action.

Examples for mental states dynamics actions are adding and removing beliefs/goals/messages/events to and from the belief-/goal-/message-/event-base, or processing instantiated plans. Note that this also includes querying the different components of the mental states, because querying always has an effect at least in the generation of an result, which also becomes a part of the mind of an agent.

As suggested, a general trace should be defined as a sequence of percepts, mental states dynamics actions and actions:

Definition 32 (general trace) *Each sequence*

$$\langle P_1, MD_1, a_1\rangle \to \langle P_2, MD_2, a_2\rangle \to \langle P_3, MD_3, a_3\rangle \to \dots$$

where each P_i is a set of percepts received at time step i, MD_i is a sequence of mental state dynamics actions, and a_i is the action scheduled for execution in the external environment, is a general trace.

Of course, based on this definition a distinct notion of equivalence can be established:

Definition 33 (general equivalence) *Two agents ag_1 and ag_2 are equivalent with respect to general state traces if they, when executed, produce the same sets of general state traces.*

Our conducted experiments and investigations showed that, when considering agents, which we expect to be similar or equivalent to a certain degree – agents which exhibited equivalent or at least similar behavior – do not generate identical general traces. This makes it necessary to refine the notion of general equivalence, taking into account different components of general traces, which gives rise to other notions of equivalence.

5.1.1. Percepts Equivalence

Now, we establish a notion of equivalence which is based on the agents' capability to perceive the environment. During an agent's evolution the environment is usually perceived on a regular basis, that is in general at the beginning of each perceive-think-act cycle. This repeated querying of the environment yields percept traces, which denote the percepts an agent receives and possibly processes in its lifetime:

Definition 34 (percept trace) *Each sequence*

$$P_1 \rightarrow P_2 \rightarrow \dots$$

where each P_i is a set of percepts received at time step i, is a percept trace.

Building on top of that, we can define that two agents are equivalent if they always generate the same trace or traces of percepts during their evolution:

Definition 35 (percepts equivalence) *Two agents ag_1 and ag_2 are equivalent with respect to percepts if they, when executed, produce the same sets of percept traces.*

5.1.2. External Actions Equivalence

The previous notion of equivalence which is based on tracing percepts that agents receive while being executed needs to be complemented with a similar notion that takes into account the effectoric capabilities of agents, that is which actions agents execute one after another. We firstly consider action traces:

Definition 36 (action trace) *Each sequence*

$$a_1 \rightarrow a_2 \rightarrow \dots$$

where each a_i is an action performed at time step i, is an action trace.

Thus, while an agent evolves over time, it leaves a trace of actions which are being performed in the external environment. Now, two agents are equivalent when they have the same traces:

Definition 37 (external actions equivalence) *Two agents ag_1 and ag_2 are equivalent with respect to external actions if they, when executed, produce the same sets of action traces.*

5.1.3. Black Box Equivalence

One of our previous definitions of equivalence made use of the agents' sensory capabilities and compared those agents by comparing their perception traces. Another definition employed the agents' effectoric capabilities and established a notion of equivalence based on which actions the agents perform in an environment. It is obvious that both definitions are complementary to one another. We are about to refine our approach by merging both definitions. Thus, agents are equivalent if processing the same percepts would yield the same actions. This way we consider each agent as a black box by neglecting its internal processes for the time being. Again, we define traces:

Definition 38 (black box trace) *Each sequence*

$$\langle P_1, a_1 \rangle \rightarrow \langle P_2, A_2 \rangle \rightarrow \langle P_3, a_4 \rangle \rightarrow \ldots$$

where each P_i is a set of percepts received at time step i and a_i is the action scheduled for execution in the external environment, is a black box trace.

Of course, based on this definition a distinct notion of equivalence can be established:

Definition 39 (black box equivalence) *Two agents ag_1 and ag_2 are equivalent with respect to black box traces if they, when executed, produce the same sets of general state traces.*

In summary, two agents are black box equivalent, if they, when executed, perceive the same data and then perform the very same actions. Our first definition on the other hand, which also takes mental state dynamics actions into account, can be considered to be a white box approach because it reveals the agents' internals.

5.1.4. Mental State Actions Equivalence

The drawbacks for the previous definitions is the fact that they are based on agents' interaction with an environment. Obviously this does not allow for comparison of agents that are not situated, rendering it necessary to come up with a notion of equivalence that can be used in such a case. It is natural to now define traces that take into account the internals of an agent:

Definition 40 (mental state actions trace) *Each sequence*

$$MD_1 \rightarrow MD_2 \rightarrow MD_3 \rightarrow \ldots$$

where MD_i is a set of mental state dynamics actions, is a mental state actions trace.

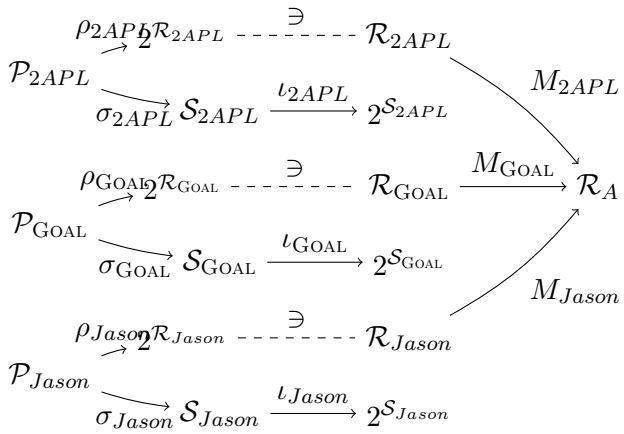


Figure 5.1.: The different data-structures that we introduce in order to compare agent-programs, and the mappings between them.

Building on top of that, another notion of agent trace equivalence can be established:

Definition 41 (mental state actions equivalence) *Two agents ag_1 and ag_2 are equivalent with mental state actions traces if they, when executed, produce the same sets of mental state actions traces.*

Expressed in prose, this means that two agents are equivalent if they perform the same operations on their mental states.

5.2. Equivalence Notions for 2APL, GOAL and Jason

In the previous section, we have elaborated on different notions of agent equivalence based on different agent traces. Now, we move away from that generic level we have assumed and move towards a more specific one. We inspect state traces of 2APL/GOAL/*Jason* agents and formulate several specialized notions of equivalence taking into account differences and similarities. It is our designated goal to compare agent programs that are implemented by means of different agent programming languages. To reach that goal it is necessary to define a *equivalence notion*, which in turn requires that we firstly examine a set of programming platforms and find out what they have in common, in order to establish a basis for the equivalence notion. We examine and compare 2APL, GOAL and *Jason* and specify the basis for equivalence by means of generic agent states. Although the considered platforms are expected to differ significantly when it comes to mental attitudes, it is also expected that a definition of generic agent states can be established without much effort. On top of these generic agent states we define generic agent runs (sequences of agent states). After that, we define a notion of agent program equivalence that is based on these generic runs. We also provide mappings from platform-specific agent runs to generic agent runs (as depicted in Fig. 5.1).

5.2.1. Agent Programs and Agent States

The first required notion is the notion of an *agent state*. We now introduce the different definitions of agent states for 2APL, GOAL and *Jason*. Additionally, we give a hint on how agent states change over time. Note, however, that it is not in the scope of this thesis to give a full description of agent syntax and semantics. We refer to the literature [23, 35, 19] for complete descriptions. We conclude this part of the thesis with a brief comparison.

The state of a 2APL agent consists of a *belief base* that is expressed by means of a Prolog program, a list of declarative goals, that constitutes the *goal base*, a set of plan entries, that constitutes the *plan base*, and the *event base*, that consists of external events received from the environment, failed plans, and received messages. Formally [23] this means:

Definition 42 (2APL agent state)
The tuple $A_\iota := \langle \iota, \sigma, \gamma, \Pi, \Theta, \xi \rangle$ *is an 2APL agent state with:*

- ι *a string representing the agent's identifier,*
- σ *a set of belief expressions constituting the belief base,*
- γ *a list of goal expressions constituting the goal base,*
- Π *a set of plan entries,*
- Θ *a ground substitution that binds domain variables to ground terms, and*
- $\xi := \langle E, I, M \rangle$ *an event-base with* E *the set of events received from external environments,* I *set of plan identifiers denoting failed plans, and* M *the set of messages sent to an agent.*

The state of a GOAL agent on the other hand consists of a knowledge base and a *belief base*, both expressed by means of a knowledge representation (KR) language[1], and a *goal base*, again expressed by the same language. Additionally it contains the current percepts, rules for action selection, received messages and actions to be performed. Formally this is:

Definition 43 (Goal agent state)
A GOAL *agent state consists of*

- *a mental state* $\langle \mathcal{D}, \Sigma, \Gamma \rangle$*, where* $\mathcal{D}$ *is called a knowledge base,* Σ *is a belief base, and* Γ *is a goal base,*
- *a set AR of* action rules *that facilitate the action selection mechanism,*
- *a set* P*of percepts representing the percepts received from the environment,*

[1] Although GOAL does not restrict the agent developer to using a specific KR language, we stick to using Prolog.

- *a set of M messages received from other agents, and*
- *a set of A actions to be executed by the environment.*

The state of a *Jason* agent, however, consists of a belief base expressed by means of a Prolog-like KR language, a plan base, a set of intentions consisting of partially instantiated plans, an event list, a set of actions to be performed, a message box for communicating, and a set of suspended intentions. Formally [17]:

Definition 44 (*Jason* agent state)
The tuple $\langle ag, C, M, T, s\rangle$ is a Jason agent state with:

- *ag is an agent program, which is specified by a set of beliefs and a set of plans,*
- *C is the circumstance, that is a tuple $\langle I, E, A\rangle$ where I is a set of intentions, each one is a stack of partially instantiated plans, E is a set of events, and A is a set of actions to be performed,*
- *$M := \langle In, Out, SI\rangle$ is a tuple where In is the mail inbox, Out is the mail outbox, and SI is the set of suspended intentions,*
- *$T := \langle R, Ap, \iota, \epsilon, \rho\rangle$ stores temporary information, R is the set of relevant plans, Ap is the set of applicable plans, ι, ϵ, and ρ keep record of a particular intention, event and applicable plan being considered during the executions, and*
- *s indicates the current step of the agent's deliberation cycle, that is processing a message, selecting an event, retrieving all relevant plans, retrieving all applicable plans, selecting one applicable plan, adding the new intended means to the set of intentions, selecting an intention, executing the selected intention or clearing an intention*

For the sake of convenience, we use the following definitions for the rest of our thesis:

Definition 45 (agent programs, agent states)

- *$\mathcal{P}_{2APL}$ is the set of 2APL agent-programs,*
- *$\mathcal{P}_{\text{GOAL}}$ is the set of* GOAL *agent-programs,*
- *$\mathcal{P}_{Jason}$ is the set of Jason agent-programs,*
- *$\mathcal{S}_{2APL}$ is the set of 2APL agent-states,*
- *$\mathcal{S}_{\text{GOAL}}$ is the set of* GOAL *agent-states, and*
- *$\mathcal{S}_{Jason}$ is the set of Jason agent-states.*

Now, we have to briefly compare the different notions of agent states in order to define a notion of generic ones. On one hand, the belief bases in 2APL and GOAL are full Prolog, the belief-base in *Jason* is logic-programming-like, consisting of facts, rules and strong negation. On the other hand, the goal-bases in 2APL and GOAL are declarative. The goal base in 2APL, is an ordered collection of goal-expressions, where every goal-expression is a conjunction of ground atoms. In GOAL the goal base is a set of goals, where each goal is either a literal or a conjunction of literals. In *Jason* there is no explicit declarative goal base. Goals, that is achievement- and test-goals, are either stored in the event base together with other events, or they are stored in the triggering-events of the instantiated plans in the agent's set of intentions. As we show later, *Jason* goals can be made explicit. We cannot compare the plan-libraries straight away, because in 2APL and *Jason* the semantics is different, and because GOAL lacks plans. For the same reasons, we do not compare intentions. Also, we do not use events, since the notion of events is different in 2APL and *Jason*, and because this notion is absent from GOAL. In summary, we restrict ourselves to using only goals and beliefs, because this is something that all three platforms have in common. Furthermore we have to make an assumption about the belief base. That is, we are going to use only the facts from the belief-bases, ignoring rules and strong negation. When it comes to goals we restrict ourselves to goal-bases that consist of a set of goals, where each goal is an atom.

5.2.2. Mental-State Trace Equivalence

An *agent run* is usually defined as a sequence of agent states. We have already defined states of 2APL, GOAL and *Jason* agents. On top of that we now elaborate on how such states are transformed by the respective interpreter. Again it should be noted that a full definition of agent program syntax and semantics for all three APL platforms is provided by the literature [23, 35, 19]. Usually an agent has an initial state, which is determined by the respective agent program. For each individual agent its agent state is transformed by applying the agent interpreter function which is implemented by the respective APL platform. We only give a brief but sufficient definition of the semantics. Formally:

Definition 46 (initial agent states)

- $\sigma_{2APL} : \mathcal{P}_{2APL} \rightarrow \mathcal{S}_{2APL}$ *maps all 2APL agent programs to their respective initial agent-states,*
- $\sigma_{\text{GOAL}} : \mathcal{P}_{\text{GOAL}} \rightarrow \mathcal{S}_{\text{GOAL}}$ *maps all* GOAL *agent programs to their respective initial agent-states,*
- $\sigma_{Jason} : \mathcal{P}_{Jason} \rightarrow \mathcal{S}_{Jason}$ *maps all Jason agent programs to their respective initial agent-states.*

2APL agent states evolve as follows: 1. instantiating plans while taking into account the goal base and the belief base, 2. executing the first action of all instantiated plans,

and 3. processing internal/external events and messages, which yields new instantiated plans. The evolution of GOAL agents, on the other hand, is facilitated by a simple instance of a sense-plan-act-cycle: 1. storing percepts and incoming messages in the belief base, and 2. randomly selecting an applicable rule and executing it, which yields actions to be executed in the environment, and 3. execute the actions. *Jason* agents are executed as follows: 1. processing percepts and incoming messages, 2. selecting an event and instantiating a plan from that event, and 3. selecting an instantiated plan and executing its first action. Formally the agents evolve by applying the respective interpreter functions:

Definition 47 (agent interpreter functions)

- $\iota_{2APL} : \mathcal{S}_{2APL} \to 2^{\mathcal{S}_{2APL}}$ *is the 2APL interpreter-function,*
- $\iota_{\text{GOAL}} : \mathcal{S}_{\text{GOAL}} \to 2^{\mathcal{S}_{\text{GOAL}}}$ *is the* GOAL *interpreter-function, and*
- $\iota_{Jason} : \mathcal{S}_{Jason} \to 2^{\mathcal{S}_{Jason}}$ *is the Jason interpreter-function.*

Agent runs are generated by repeatedly applying interpreter-functions:

Definition 48 (agent runs)

- $\mathcal{R}_{2APL} := (S_{2APL})^+$ *is the set of 2APL-runs,*
- $\mathcal{R}_{\text{GOAL}} := (S_{\text{GOAL}})^+$ *is the set of* GOAL*-runs,*
- $\mathcal{R}_{Jason} := (S_{Jason})^+$ *is the set of Jason-runs,*
- $\rho_{2APL} : \mathcal{P}_{2APL} \to 2^{\mathcal{R}_{2APL}}$ *is the function that computes all 2APL-runs of a 2APL agent-program, using* ι_{2APL} *and the initial agent-state derived from a given agent-program,*
- $\rho_{\text{GOAL}} : \mathcal{P}_{\text{GOAL}} \to 2^{\mathcal{R}_{\text{GOAL}}}$ *is the function that computes all* GOAL*-runs of a* GOAL *agent-program, using* ι_{GOAL} *and the initial agent-state derived from a given agent-program, and*
- $\rho_{Jason} : \mathcal{P}_{Jason} \to 2^{\mathcal{R}_{Jason}}$ *is the function that computes all Jason-runs of a Jason agent-program, using* ι_{Jason} *and the initial agent-state derived from a given agent-program.*

Note, that for the sake of abstraction, we assume that the interpreter functions are *deterministic*. Non-determinism would be absolutely feasible for the framework that we are about to introduce, but for supporting the readability of this thesis, we refrain from coping with that specific issue now.

5.2.3. Generic Agent States and Agent Runs

In the previous subsections, we have shown how 2APL-, GOAL- and *Jason*-agents are defined and how they evolve during runtime. We have also calculated that, if we chose to define generic agent states and runs, judging from the notions that all three platforms have in common, it would make sense to use primitive (that is consisting of ground literals) beliefs and goals as the main building block for defining generic agent states:

Definition 49 (generic agent state)

- $B_A := \{b_1, b_2, \ldots\}$ *is a set of generic beliefs,*
- $G_A := \{g_1, g_2, \ldots\}$ *is a set of generic goals, and*
- $\mathcal{S}_A$ *is the set of generic agent states where each* $s_i \in \mathcal{S}$ *is a tuple* $\langle B_i, G_i \rangle$*, where* $B_i \subseteq B_A$ *is a set of beliefs and* $G_i \subseteq G_A$ *is a set of generic goals.*

Of course, a generic agent run is a sequence of generic agent states:

Definition 50 (generic agent run)

- *Each sequence* $\langle B_0, G_0 \rangle \rightarrow \langle B_1, G_1 \rangle \ldots$ *is an abstract agent run, and*
- $\mathcal{R}_A := (S_A)^+$ *is the set of all abstract runs.*

Now we have to define a set of generalization mappings, that map specific agent states and runs to generic ones:

Definition 51 (generalization mappings)

- $\mu_{2APL} : \mathcal{S}_{2APL} \rightarrow \mathcal{S}_A$ *maps each 2APL agent-state to an abstract agent-state,*
- $\mu_{\text{GOAL}} : \mathcal{S}_{\text{GOAL}} \rightarrow \mathcal{S}_A$ *maps each* GOAL *agent-state to an abstract agent-state,*
- $\mu_{Jason} : \mathcal{S}_{Jason} \rightarrow \mathcal{S}_A$ *maps each Jason agent-state to an abstract agent-state,*
- $M_{2APL} : \mathcal{R}_{2APL} \rightarrow \mathcal{R}_A$ *with* $M_{2APL} : (s_1, \ldots, s_n) \mapsto (\mu_{2APL}(s_1), \ldots, \mu_{2APL}(s_n))$ *maps each 2APL agent run to an abstract agent run,*
- $M_{\text{GOAL}} : \mathcal{R}_{\text{GOAL}} \rightarrow \mathcal{R}_A$ *with* $M_{\text{GOAL}} : (s_1, \ldots, s_n) \mapsto (\mu_{\text{GOAL}}(s_1), \ldots, \mu_{\text{GOAL}}(s_n))$ *maps each* GOAL *agent run to an abstract agent run, and*
- $M_{Jason} : \mathcal{R}_{Jason} \rightarrow \mathcal{R}_A$ *with* $M_{Jason} : (s_1, \ldots, s_n) \mapsto (\mu_{Jason}(s_1), \ldots, \mu_{Jason}(s_n))$ *maps each Jason agent run to an abstract agent run.*

We do not have to elaborate the mappings from specific agent runs to generic ones, as these are inductive in nature. The mappings from specific agent states to generic ones, however, are more interesting. For 2APL we consider the belief base and copy each fact contained therein to the generic belief-base. An equivalent procedure is applied to all atomic goals from the goal base. The procedure for mapping GOAL's mental attitudes is

the same. Again we keep all facts from the belief base, while ignoring the rules, and copy all atomic goals from the goal base to the generic one. For *Jason* mapping the beliefs is almost the same, except for the facts being stripped off their annotations. Because *Jason* does not hold a notion of declarative beliefs, we have to apply a special treatment to the mental attitudes. Goals can be extracted 1. from the event-base, and 2. from the triggering-events of instantiated plans.

5.2.4. Agent State and Agent Run Similarity

Now, with the definitions of generic agent states and generic agent runs in place, we can concentrate on *agent state* and *agent run equivalence*. We expect repeating states for two reasons: 1. the mapping from specific agent states to generic ones, which acts as a kind of filtering-function, might yield for two different specific agent states the same generic one, and 2. agent states in general might repeat under certain conditions (e.g. when nothing happens). In order not to burden ourselves with repeating states we introduce a compression function:

Definition 52 (compression function)
$\delta : (e_1, e_2, \ldots, e_n) \mapsto (e'_1, e'_2, \ldots, e'_m)$ *is the compression function that maps each sequence* $s := (a_1, \ldots, a_n)$ *defined over an arbitrary set* A *to a second one* $s' := (a'_1, a'_2, \ldots, a'_m)$ *where* s' *is* s *without repeated entries* [2].

In order to reason about generic agent runs effectively, we define a couple of filtering projections that allow us to restrict the considered beliefs and goals to subsets. Sometimes it might not be necessary to take the full belief and goal bases into account:

Definition 53 (filtering projections)

- $\pi_B : \mathcal{S}_A \to B$ *with* $\pi_B : \langle B, G\rangle \mapsto B$ *projects an abstract agent-state to the respective beliefs,*
- $\pi_G : \mathcal{S}_A \to G$ *with* $\pi_B : \langle B, G\rangle \mapsto G$ *projects an abstract agent-state to the respective goals,*
- $\Pi_B : \mathcal{R}_A \to B^*$ *with*

  $$\Pi_B : (s_1, s_2, \ldots) \mapsto (\pi_B(s_1), \pi_B(s_2), \ldots)$$

  projects all abstract agent-runs to sequences of beliefs, and
- $\Pi_G : \mathcal{R}_A \to G^*$ *with*

  $$\Pi_G : (s_1, s_2, \ldots) \mapsto (\pi_G(s_1), \pi_G(s_2), \ldots)$$

  projects all abstract agent-runs to sequences of goals.

[2] Example: $\delta(1, 2, 2, 3, 1, 1) = (1, 2, 3, 1)$

[3]

It is about time to put things together and define the desired notion of equivalence:

Definition 54 (n-B-/n-G-/n-BG-equivalent)

- *two agent-programs* p_{l_1}, p_{l_2} *with* $l_1, l_2 \in \{\texttt{2APL}, \textsc{Goal}, \texttt{Jason}\}$ *and* $p_1 \in \mathcal{P}_{l_1}, p_2 \in \mathcal{P}_{l_2}$ *are* n-B-equivalent *if*

$$r_1 := \delta(\pi_B(\mu_{l_1}\rho(p_1))) = \delta(\pi_B(\mu_{l_2}\rho(p_2))) =: r_2 \ \wedge \\ \mid r_1 \mid = n = \mid r_2 \mid$$

- *two agent-programs* p_{l_1}, p_{l_2} *with* $l_1, l_2 \in \{\texttt{2APL}, \textsc{Goal}, \texttt{Jason}\}$ *and* $p_1 \in \mathcal{P}_{l_1}, p_2 \in \mathcal{P}_{l_2}$ *are* n-G-equivalent *if*

$$r_1 := \delta(\pi_G(\mu_{l_1}\rho(p_1))) = \delta(\pi_G(\mu_{l_2}\rho(p_2))) =: r_2 \ \wedge \\ \mid r_1 \mid = n = \mid r_2 \mid$$

- *two agent-programs* p_{l_1}, p_{l_2} *with* $l_1, l_2 \in \{\texttt{2APL}, \textsc{Goal}, \texttt{Jason}\}$ *and* $p_1 \in \mathcal{P}_{l_1}, p_2 \in \mathcal{P}_{l_2}$ *are* n-BG-equivalent *if*

$$r_1 := \delta(\mu_{l_1}\rho(p_1)) = \delta(\mu_{l_2}\rho(p_2)) =: r_2 \wedge \ \mid r_1 \mid = n = \mid r_2 \mid$$

5.3. Summary

In this chapter, we have focused on two topics. Firstly we have defined and elaborated on several general notions of agent equivalence. These notions took into account traces that agents generate in their lifetime and concentrated on different components of such traces. After that we have established notions of equivalence based on a comparison of 2APL, Goal and *Jason* in our second step.

[3] Example: $\Pi_{\{fib(1,1)\}}(\langle\{fib(1,1)., fib(2,1).\}, \emptyset\rangle)$ is $(\langle\{fib(1,1).\}, \emptyset\rangle)$.

6. APLEIS: An Environment Interface Standard for Agent-Oriented Programming

Earlier, in Chapter 4, we have provided a formal definition of the kind of multi-agent systems that we are interested in. As a reminder, we concern ourselves with heterogeneous multi-agent systems, that is multi-agent systems that are populated by agents which are implemented in and executed by different agent platforms. We assume a strict separation of concerns that discerns agents as the vessels for intelligence and entities that provide effectoric and sensory capabilities.

In this and in the following chapter we approach the formalized architecture in a practical manner. We now concentrate on the layer that can be identified between agents and agent platforms on one side and environments on the other one. The EIS (environment interface standard) for APL [12] is a proposed standard for agent/platform-environment interaction. It is motivated by the following observation. In general, considering different agent platforms, it is an effort to implement an environment or to connect to an already existing one. Different platforms, of course, have different means to do so, but although the approaches differ, they also have similarities that can be exploited. Providing a standard for connecting to environments is expected to lead to an overall reduction of the amount of work and time that has to be invested. Each platform that supports such a standard would ideally be capable of connecting to any environment that is compliant to that standard.

The design of the EIS implementation requires three distinct steps:

1. extracting *generalized interface principles* from a set of examined agent platforms, in our case 2APL, Goal, and *Jason*,

2. building a *interface meta-model* on top of the principles, that reuses as much as possible, while being as general as possible, and

3. implementing an *EIS-package* for a specific programming language, in our case for Java[1].

In this chapter, we firstly elaborate on the generalized interface principles. Secondly, we explain the interface meta-model. After that, we heavily concentrate on the details of the interface. This includes a definition of an intermediate language for facilitating

[1] Of course, any other programming language can be used.

communication between agents and environments, an agents-entities system that establishes situatedness, an environment management system for execution control, and an environment querying system for statistics.

6.1. Interface Principles and Interface Meta-Model

Our examinations and insights gained by the earlier comparison of the APL platforms yield a set of seven distinct interface principles, which are the basis for the interface meta-model. These principles are derived from two sources. The first one comes from a comparison on how different platforms connect to environment, and the second one comes from the meta-goals that we have associated with this work.

The interface principles are as follows:

1. **Portability:** Exchanging environments between platforms should be easy. That is, if an environment has been written while experimenting with a specific platform, EIS should facilitate that the environment can easily be distributed. To that end, EIS should help wrapping the environment or a connector to an environment into a package, in our case a JAR-file[2], which then can easily be plugged into a different platform.

2. **Environment interface generality:** We impose one minimal restriction. We assume that EIS does not provide any means for scheduling actions. Actions are either scheduled for execution by the platform or by the environment model. EIS, which only acts as an interface between platforms and environments, should not have any control here. Furthermore, there are no assumptions about agent coordination. The responsibility for coordinating agents lies either on the platform side and is established by a middleware (for example communication middlewares like Jade [16]), or it is facilitated by constructs in the environment-model (for example artifacts [53]). Then, we do not assume anything about what is controllable in an environment, except for the notion of so called *controllable entities*, which we introduce later and which provides sensory and effectoric capabilities to agents. Also, there are no assumptions about how agent/platforms control these entities. Finally, we do not require anything about which technical option has to be used to connect to the specific environment. That is, the developer is free to decide if he wants to connect via TCP/IP, JNI et cetera.

3. **Separation of concerns:** It is assumed that the agents are separated from the environment. Thus, we oppose all approaches that define agents as environmental properties that are part of the environment. We assess that there is a conceptual gap between agents, which are percept-processors and action-generators, on the platform side and *controllable entities*, which analogously are percept-generators and action-processors, on the environment side. This gap can, in general, be

[2]JAR-files are **J**ava **AR**chives generally used to distribute applications. JAR-files usually contain compiled classes and resources, e.g. media files. JAR-files can also be loaded during runtime.

arbitrary in nature. Thus we cannot assume that, from a software engineering perspective, the agents, which are objects, are always a part of the environment, which is also an object. Also we cannot assume anything about the structure of agents and controllable entities, because, again, we want to be as general as possible. This implies that EIS, in its role as a glue layer between agents and platforms on one side and controllable entities on the other, cannot and should not store any high-level data about either agents or controllable entities, except for strings that act as unique identifiers.

4. **Unified connections:** Clearly these principles yield a meta-model consisting of several, specialized components. We assume that the connections between these components should not be restrictive. This includes different notions of sensing. Active sensing, that is sensing by executing special sensing-actions, and passive sensing, that is sensing as notifications by the environment, should be supported and facilitated by EIS. It is assumed that acting is facilitated as well. In order to stay in line with the previously defined principle it is necessary to provide means for associating agents and controllable entities. This association is called the *agents-entities-relation* and is assumed to be dynamic. All the functionality of these unified connections is assumed to be established by providing a set of specialized interface methods.

5. **Standards for actions/percepts and similar notions:** Expecting a wide variety of environments to connect to and a wide variety of platforms to connect from, it is necessary to provide a convention for the communication between the different components. Since different platforms come with different knowledge-representation languages it is essential to define a compromise, that can be used by as many platforms as possible. It is important to ensure a high expressiveness, together with a simplicity in usage.

6. **Support for heterogeneity:** EIS, in its function as a glue-layer between agents and platforms on one side and environments on the other, should be responsible for establishing heterogeneity. Heterogeneity means connecting agents from different platforms at one and the same instance of an environment at the same time.

7. **Means for data acquisition:** This principle is in line with the overall goal of this thesis, that is comparing APL platforms. It is an addition to the original design and is supposed to facilitate querying the state of an environment or the respective environment interface in order to acquire useful data that can be evaluated. EIS should provide means for doing so.

After discussing the principles, we elaborate on the interface meta-model that is built on top of these very principles. The meta-model consists of a couple of components that constitute the EIS implementation, and a communication language that facilitates the cooperation of the components. Figure 6.1 gives a brief overview of the components and their channels of interaction. We differentiate between three layers with varying degrees of our concern. The *APL platform side* contains the platform and the agents,

the *environment side* contains the environment and the *interface layer*, which connects platforms/agents and environments in a generalized fashion. Note, that we abstract both the APL platform side and the environment side as much as possible and concentrate on the details of the interface layer. Because of the wide design-space for implementing platforms/agents/environments, we cannot afford to assume that much, as we have already discussed in the interface principles.

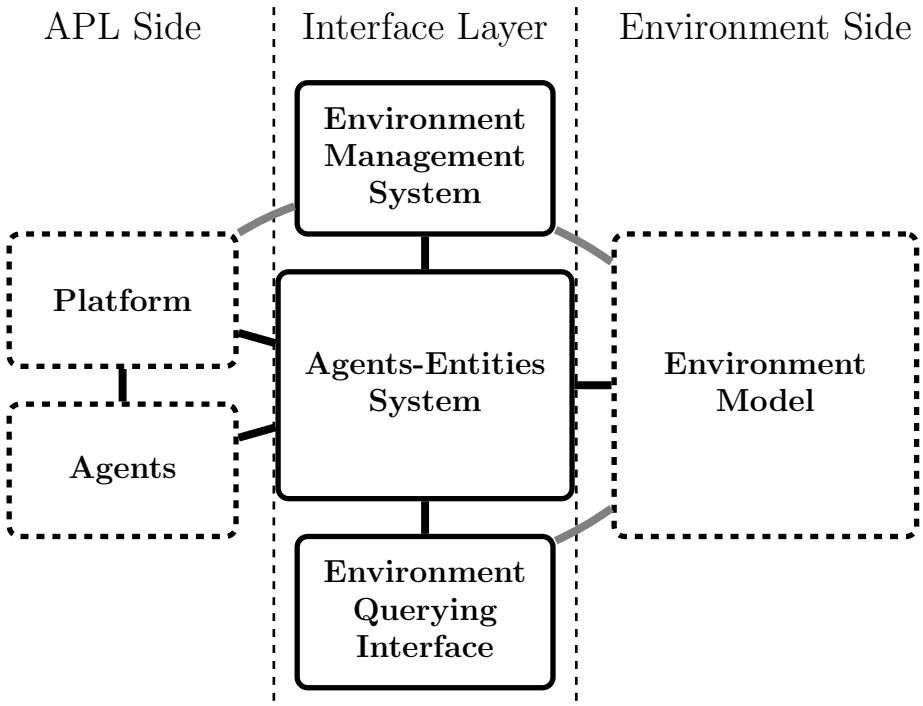


Figure 6.1.: The interface meta-model. We have three different layers. The platform and its agents live on the APL side. The environment management system, that facilitates the manipulation of the environments executional state, the agents-entities system, that connects agents to controllable entities, and the environment-querying system, that allows for querying the environment and the environment interface for data, all live on the interface layer. The environment model lives on the environment side.

In the following, we discuss an interface meta-model that can be derived from the recently introduced interface principles. We discuss the components first, in an order of increasing concern, that is, starting with those components that we do not assume that much about, and secondly the interaction language that facilitates the communication between several components. The interface meta model then leads to the implementation.

1. **Platform:** We do not assume anything about the internal details of a platform,

which we define as any structure that manages and executes agents.

2. **Agents:** Again, we do not assume that much about the internal details of an agent. We only assume that an agent is any structure that conforms to Russel-Norvig's definition [55].

3. **Environment model:** Here we require a little more. We have two assumptions: 1. the environment contains *controllable entities*, which are entities that agents can connect to in order to act and perceive in the environment, and 2. the environment has an execution state that can be queried and manipulated. Note, that the second requirement is not obligatory, whereas the first one is.

4. **Agents-entities system:** We adhere to the notion that there are agents on the agent platform side and controllable entities on the environment side. We assume that agents are percept processors and action generators, and we assume that controllable entities are action processors and percept generators. Agents are capable of querying the (partial) state of the environment through the controllable entities. We call this active sensing. Additionally, controllable entities can send the agents percepts as notifications without the agents requesting it. Finally, agents can manipulate the state of the environment through the controllable entities, sensing and acting are established all by means provided by the controllable entities. Percepts and actions are defined in terms of a components interaction language, that we are going to present soon.

   In order to establish the situatedness of an agent, it has to be *connected* to at least one controllable entity. This connection is facilitated by the so called *agents-entities relation*. In the agents-entities system we represent both the sets of agents and controllable entities by a set of identifiers. We only work with identifiers, because this would ensure that we do not store any high-level data about the agents and controllable entities, which are platform and environmental properties respectively.

   The agents-entities relation is in general any subset of the Cartesian product of the set of agent identifiers and the set of controllable entity identifiers. We assume that this is an arbitrary relation, but usually this has to be a one-to-one mapping. Figure 6.2 depicts the agents-entities relation and summarizes the agents-entities system.

5. **Environment management system:** This component is intended to allow for the manipulation of the executional state of the environment by connected components. We assume that the environment can have several distinct states: 1. `initialized` denotes that the environment interface is instantiated, this instantiation can be accompanied by a couple of initialization parameters, 2. `paused` denotes that the environment is paused, this state is reached once the connection to the environment has been established after the initialization of the environment interface, 3. `started` denotes that the environment is running and that the entities

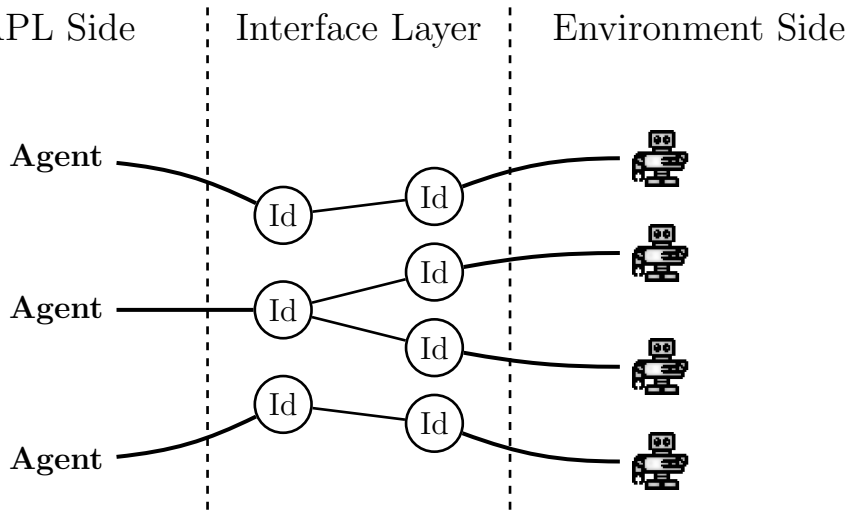


Figure 6.2.: The agent entities-relation. Agents are percept processors and action generators that live on the APL side. Controllable entities are percept generators and action processors and live on the environment side. Both agents and controllable entities are represented via identifiers on the interface layer.

generate percepts and process actions, and 4. `killed` denotes that the environment and all the resources of the environment interface can be released.

The state transitions are as follows:

a) from `initialized` to `initialized` when an `INIT`-command is sent, which can carry a set of parameters encoded as a set of key-value-pairs,

b) from `initialized` to `paused` when the connection to the environment is established,

c) from `paused` to `started` when a `START`-command is sent,

d) from `started` to `paused` when a `PAUSE`-command is sent, and finally

e) from `started` to `killed` and from `paused` to `killed` when a `KILL`-command is sent.

Every time a state-change occurs it is assumed that the components on the APL-side are notified. Figure 6.3 depicts and summarizes the environment management system.

6. **Environment querying interface:** This component allows for querying the state of the environment and the environment interface in order to retrieve useful data that might be used for statistics, profiling and debugging.

7. **Interface intermediate language:** This is the language for inter-component communication. The language is intended to facilitate the interaction of agents

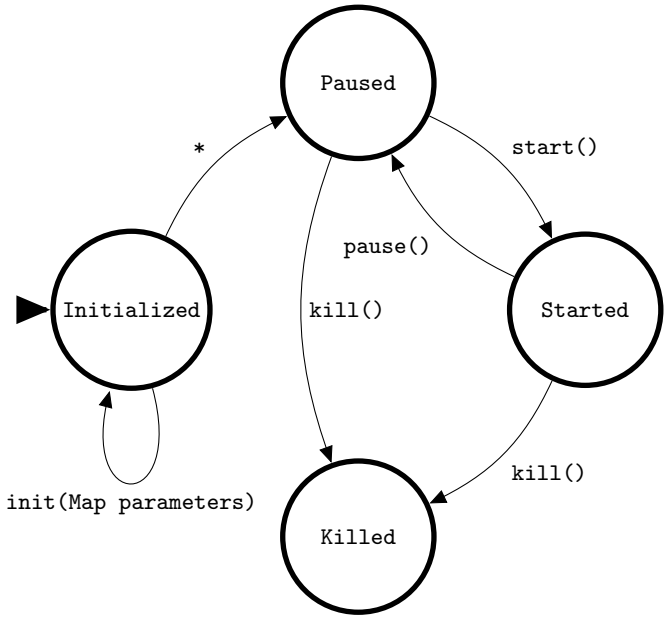


Figure 6.3.: The different states of the environment management system. The environment interface is initialized once it is instantiated. A soon as the connection to the environment is established the environment interface is paused. When it is started the controllable entities process actions and generate percepts. Once it is killed it is in the killed state.

with the environment and provides means for expressing tokens of data on an adequate level of expressiveness. This is a kind of convention that constitutes a contract between users of an interface and the interface itself. This corresponds to the standards for actions/percepts-principle.

6.2. Using Listeners

We now focus on the exact correspondence between an environment-interface and the components. In general, we allow for a two-way connection via *interactions* that are performed by the components and *notifications* that are performed by the environment-interface.

Interactions are facilitated by function calls to the environment-interface, that can yield a return-value. For notifications we employ the *observer design pattern* (call-back methods, known as *listeners* in Java). The observer pattern defines that a *subject* maintains a list of *observers*. The subject informs the observers of any state change by calling one of their methods. This way distributed event handling is facilitated. The observer pattern is usually employed when a state-change in one object requires changing another one. This is the reason why we made that choice. The subject in the observer pattern usually provides functionality for *attaching* and *detaching* observers, and for

notifying all attached observers. The observer, on the other hand, defines an *updating* interface to receive update notification from the subject.

We allow for both interactions and notifications, because this approach is the least restrictive one. This clearly corresponds to the notions of *polling* (an agent performs an action to query the state of the environment) and *interrupts* (the environment sends percepts to the agents as in the Contest).

While we investigate the interactions with the specific components later we now consider managing the listeners. EIS allows for two types of listeners: *environment listeners* are employed for communicating data about the environment and the environment interface, *agent listeners*, on the other hand, are used to communicate data about changes with respect to the entities and agents-entities-relation. This is the set of methods that correspond to the listener functionality:

- `attachEnvironmentListener(EnvironmentListener listener)` registers an environment listener with the environment interface.
- `detachEnvironmentListener(EnvironmentListener listener)` unregisters an environment listener from the environment interface.
- `attachAgentListener(String agent, AgentListener listener)` registers an agent listener with the environment interface.
- `detachAgentListener(String agent, AgentListener listener)` unregisters an agent listener from the environment interface.

6.3. Interface Intermediate Language

The cooperation between the different involved components is partly established by calling specific methods in order to trigger events, and partly by transmitting tokens that contain useful data. The *interface intermediate language* (IILang) is a manageable language definition for actions and percepts with an adequate level of expressivity.

The Interface Intermediate Language (IILang) consists of 1. *data containers* (for example actions and percepts), and 2. *parameters* to those containers (see Figure 6.4). Members of the IILang are stored as abstract syntax trees formed by Java-objects, which can be printed either in an XML- or in a logic-programming style for the sake of readability.

Parameters are either identifiers, numbers, truth-values, functions over parameters, or lists of parameters. These are the Java-classes representing parameters:

- `eis.iilang.Identifier` represents an identifier,
- `eis.iilang.Numeral` represents a number,
- `eis.iilang.TruthValue` represents a truth-value,
- `eis.iilang.Function` represents a function over parameters, and

- `eis.iilang.ParameterList` represents a list of parameters.

Data containers, however, are 1. actions that are performed by agents via entities, and 2. percepts sent by the environment-interface and received by agents or percepts as results of actions. Each of these data containers consists of a name, and a set of parameters. Here are the respective classes:

- `eis.iilang.Action` represents an action.
- `eis.iilang.Percept` is a percept.

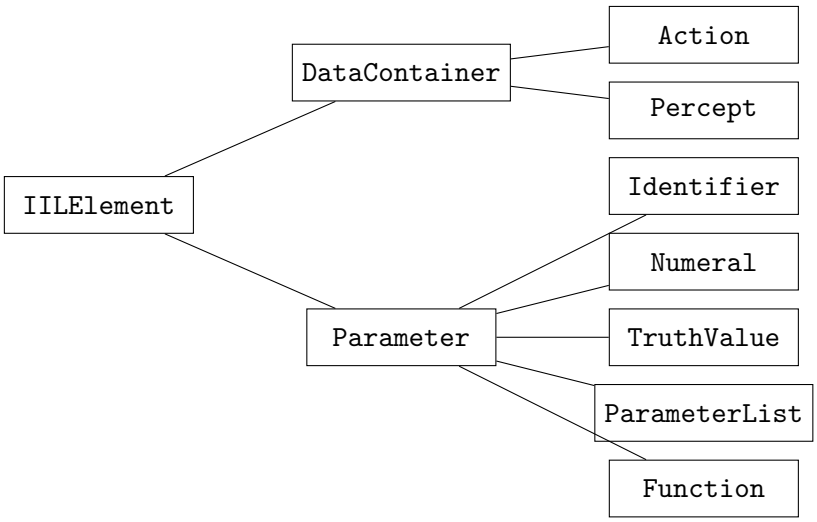


Figure 6.4.: The inheritance relation of the IIL-elements. Actions and percepts are data-containers. Each data-container consists of a name and an ordered collection of parameters. Each parameter is either an identifier, a numeral, a list of parameters or a function of parameters.

While developing a contract for agent-environment interaction our examinations led us to the conclusion, that it is necessary to impose a restriction, that is a concrete and limited syntax, in order to facilitate broad portability. The idea of using plain Java-objects as means for expressing percepts and actions sounded very appealing right at the beginning, but proved to be inadequate for our purposes. Following that path would have led to the requirement that platform developers had to customize their software for every individual environment, which is clearly a violation of our intentions. The expressiveness of our language is comparable to that of logic programming. We had Prolog in mind for a good deal of our design time, knowing that a lot can be expressed with a language that follows the logic programming paradigm for composition of literals.

We now consider less than a handful of examples.

1. The visibility of a red ball can be represented by this percept rendered as a Prolog-like string:

```
red(ball)
```

Rendered as XML the same percept would look like this:

```
<percept name="red">
  <perceptParameter>
    <identifier value="ball"/>
  </perceptParameter>
</percept>
```

2. The visibility of three nodes of a graph can look like this in the Prolog-like notation:

```
nodes([n3,n1,n4])
```

And like this rendered as XML:

```
<percept name="nodes">
  <perceptParameter>
    <parameterList>
      <identifier value="n3"/>
      <identifier value="n1"/>
      <identifier value="n4"/>
    </parameterList>
  </perceptParameter>
</percept>
```

3. And finally, an action that is intended to move an entity to a given position on a two-dimensional grid can look like this in the Prolog-like notation:

```
goto(4,2)
```

Or alternatively as XML:

```
<action name="goto">
  <actionParameter>
    <number value="4"/>
  </actionParameter>
  <actionParameter>
    <number value="2"/>
  </actionParameter>
</action>
```

6.4. Agents/Entities System

The agents/entities system (AES) is the component that connects agents to entities and facilitates acting and perceiving in the environments. Provided functionalities include managing the agents-entities-relation, support for entity-types, acting and perceiving.

6.4.1. Managing the Agents-Entities-Relation

We assume that every multi-agent system that makes use of EIS has two special sets, that is $EntId$ which contains all the valid identifiers of the entities that populate the environment and $AgId$ that contains all identifiers of the agents which are executed by the platform. The agents-entities-relation, which formally is a subset of $EntId \times AgId$, that is the cartesian product of the set of entity-identifiers and the set of agent-identifiers, is internally represented as a map of strings to strings. This set-up ensures that the environment interface does not store any special structures that are specific to the agents and entities, thus making sure that the environment interface remains agnostic to that matter. EIS provides a set of methods for manipulating the agents-entities-relation, which we present now:

- `registerAgent(String agent)` registers an agent to the environment interface, which is an absolute requirement for an agent to interact with the environment. This method raises an `AgentException` in the case that the agent has already been registered to the environment interface earlier.
- `unregisterAgent(String agent)` is the complement `registerAgent`. It unregisters an agent from the environment interface, thus disabling any interaction between the agent and the environment. An `AgentException` is thrown in the case that the agent has not been registered upon attempting to unregister the agent.
- `associateEntity(String agent, String entity)` associates an agent with an entity, thus enabling the agent to access the sensory and effectoric capabilities provided by the entity. This method yields a `RelationException` if the attempt to associate fails. This is the case if the agent has not been registered or if the entity does not exist.
- `freeEntity(String entity)` frees an entity by removing the entities relations to all its associated agents. If the execution of the method fails because the entity does not exist a `RelationException` is raised.
- `freeAgent(String agent)` is similar to `freeAgent` with the difference that the method frees an agent from all its associated entities. The method throws a `RelationException` if the agent has not been registered.
- `freePair(String agent, String entity)` renders the association between an agent and an entity defunct. A `RelationException` is raised if the agent has not been registered, the entity does not exist, or if both have not been associated.

For querying the agents-entities-relation EIS provides another set of methods:

- `getAgents()` returns the set of all agents that are registered to the environment interface, stored as an instance of `Collection<String>`.
- `getEntities()` yields the set of all entities, also stored as a `Collection<String>`.

- `getAssociatedEntities(String agent)` returns the set of all entities that are associated with a given agent. If the agent has not been registered, an `AgentException` is thrown. In the case of success the entities are stored in a `Collection<String>` which is returned.

- `getAssociatedAgents(String entity)` when called, returns an instance of the Java class `Collection<String>` which contains all the agents that are associated with a given entity. Throws an `EntityException` if the entity does not exist.

- `getFreeEntities()` returns the set of all free entities, that is all entities that are not associated with any agents. The set is returned as a `Collection<String>`.

We assume that the agents-entities-relation is subject to change that is not required to be initiated by the platform only. For the sake of conveying information about changes in the relation, we employ the well-known observer-design pattern. Components of the platform that should be informed about changes are supposed to implement the `EnvironmentListener` interface and these methods, which can be called during runtime:

- `handleFreeEntity(String entity)` is called when an entity is freed,

- `handleDeletedEntity(String entity)` is called when an entity is deleted, and

- `handleNewEntity(String entity)` is called when an entity is created.

We have intended that the set of methods for querying the agents-entities-relation described above should allow it to be automatically managed by agent platforms. That is automatically associating agents with entities, reacting to the sudden appearance of entities during runtime, and similar scenarios.

6.4.2. Entity Types

On top of the agents-entities-relation, each entity has a type that indicates its effectoric and sensory capabilities, assuming that EIS facilitates entities with different capabilities. This set-up would allow platforms to instantiate a specific agent based not only on the name of the entity under consideration, but also taking into account its type. Entity types are represented by Java strings. EIS provides a single method that gives information about the type of an entity:

- `getType(String entity)` returns a `String` representing the type of the given entity. This method yields an `EntityException` if the entity does not exist.

6.4.3. Acting

Performing actions in the environment is made possible by a single method that can be invoked:

- `performAction(String agent, Action action, String... entities)` executes an action for a given agent, while taking into account a set of associated entities. This method returns a mapping from strings to percepts or throws an instance of `ActException` if the execution fails.

Starting with the preliminary assumption that the agent which intends to perform an action is associated with a set of entities, the above method does the following: 1. checking if the attempt to perform the action is valid, 2. executing the action, and 3. returning the result of the action in the form of percepts.

The first step implies the execution of an internal validation mechanism which takes into account the action itself and the entities that are supposed to execute it. This mechanism can, in the case of concluding invalidity, yield the following results:

- An instance of `NoEnvironmentException` is thrown if the environment-interface is not connected to an environment. If there is a connection, an instance of `ActException` should be thrown.
- If the syntax of the action is wrong, which is heavily specific to the environment itself, the `ActException` has the type `WRONGSYNTAX`. That is the case when the name of the action is not available and when the parameters do not match (number of parameters or their types and structure).
- If the method invocation is equivalent to the attempt of performing an agent of an unregistered agent the exception's type is `NOTREGISTERED`.
- If the agent is registered, but it has no associated entities the type is `NOENTITIES`.
- If it is attempted to perform an action via an entity that is not associated with the agent, the exception's type is `WRONGENTITY`.
- If the action is not supported by the type of one of the associated entities the type of the exception is `NOTSUPPORTEDBYTYPE`
- And finally the type `FAILURE` indicates that the action has failed although it matched all other above requirements.

A successfully executed action is indicated by the `performAction`-method yielding a mapping that maps entities to percepts. This respects the possibility that a single agent can be associated with several entities and thus a given action can be performed by those entities. The mapping then allows us to discern the results yielded by the different agents performing one and the same action.

6.4.4. Perceiving

Perceiving is established by two mechanisms. Firstly, a method allows to retrieve all percepts that are currently available to an agent, which could be called percepts-on-request. Secondly, percepts can be delivered in the form of percepts-as-notifications, that

is the environment interface can send percepts to agents, without this being triggered by the agents themselves.

The following method establishes the percepts-on-request mechanism:

- `getAllPercepts(String agent, String... entities)` yields all percepts that are available to an agent through a subset of its associated entities. The percepts are stored in a mapping that maps strings to collections of percepts. The method throws an exception if perceiving fails.

Similar to executing an action, but a little less sophisticated, an internal mechanism firstly determines if perception is possible and/or would be successful, and secondly yields all available percepts if so. An instance of `PerceiveException` is thrown if 1. the environment is not running, 2. the agent to perceive is not registered, 3. the agent has no associated entities, and 4. not all of the entities passed as arguments to the method are associated with the agents. All these requirements are checked in the same order represented here.

It should be noted at this point, that we make use of Java's exception handling mechanism in order to enforce the implementation contract that EIS constitutes. Every violation of that very contract is indicated by raising an exception that, by its type and optional additional data, communicates where that violation has occurred.

The return value of the `getAllPercepts` is a mapping from strings representing entity-names to collections of percepts. Again, this is done in order to allow for identifying which percepts were generated by which entity.

Finally, percepts-as-notifications are made possible by the method that has to be implemented when using the `AgentListener`-interface:

- `handlePercept(String agent, Percept percept)` is called when an agent should be provided with a percept without the agent explicitly requesting so.

6.5. Environment Management System

The environment management system (EMS) is that component of EIS which is responsible for maintaining and managing the state of specific environments and environment interfaces. It is intended to provide means for querying and updating the state. Possible areas of application are pausing the environment in order to inspect its state and starting/stopping the execution of an environment during a series of test runs. Internally a finite-state machine denotes the executional states of the environment interface and possible state transitions. By the provided access to that structure, platforms gain the possibility to query the state and change it if allowed. It is assumed that some transitions are not possible due to restraints imposed by the respective environment, thus the platform must be able to get certainty about the possibilities by querying the internals.

Figure 6.3 shows the EMS with its states and state transitions. We identify the following states:

- `Initialized` denotes that the environment interface has been successfully set up and initialized. In this state the environment interface accepts initialization commands for set-up purposes.

- `Paused` represents that the connection to the environment has been successfully established. Now the environment can be started.

- `Started` indicates that the environment is running. In this state it processes actions and generates percepts. It is the only state in which this happens.

- `Killed` shows that the environment is killed and ready to have its resources released and be deallocated.

An ideal living cycle in a normal execution would be the sequence `Initialized-Paused-Started-Killed`. Usually, we expect alternations of the states `Paused` and `Started` during execution.

Besides the four essential states, we also identify the following state-transition-triggers, represented by the respective Java-methods:

- `init(Map<String,Parameter> parameters)` initializes the environment interface by providing parameters in the form of key-value pairs. Keys are Java-strings and values are instances of the `Parameter`-class, that assume the values described before.

- `start()` starts the environment interface.

- `pause()` pauses the environment interface.

- `kill()` kills the environment interface.

All of these methods throw an instance of the `ManagementException`-class if an error occurs.

The following state-transitions are in general allowed (compare with Figure 6.3):

- `Initialized` to `Initialized` via (init(...)): Every invocation of the `init(...)` method is supposed to initialize the environment interface.

- `Initialized` to `Paused` via (*): After an procedure that is responsible for establishing the connection between the environment and the environment interface, which is assumed to be some internal mechanism specific to the environment, the EMS automatically moves to the `Paused`-state.

- `Paused` to `Started` via `start()`: The execution of the environment interface is started. Actions are processed and percepts are generated.

- `Paused` to `Killed` via `kill()`: The environment interface is killed while its execution is on hold.

- `Started` to `Paused` via `pause()`: The execution of the environment interface is paused. No actions are processed and no percepts are generated.

- `Started` to `Killed` via `kill()`: The environment interface is killed while it is running.

These transitions should facilitate all required functionalities. It should be noted that although resetting an environment is not explicitly supported, it can be modeled by 1. killing the environment interface, 2. dereferencing it, 3. instantiating a new environment interface object, and 4. initializing the new object with the same parameters.

For querying the current state and for retrieving the capabilities of the EMS, EIS provides another set of methods:

- `EnvironmentState getState()` yields the currents state of the environment interface.

- `boolean isInitSupported()` returns `true` if initializing is supported, `false` otherwise.

- `boolean isStartSupported()` returns `true` if starting is supported, `false` otherwise.

- `boolean isPauseSupported()` returns `true` if pausing is supported, `false` otherwise.

- `boolean isKillSupported()` returns `true` if killing is supported, `false` otherwise.

Different definitions of the state transition function are possible. These are restricted only by the following assumptions[3]:

- As soon as an environment interface has been instantiated it is assumed to be in the `Initialized`-state.

- As soon as the interface has established its connection to the environment it is in the `Paused`-state.

- Only when the interface is in the `Killed`-state it and its resources can be released.

- Every time a state-transition occurs, the environment interface notifies the registered environment-listeners.

[3]This has changed in the meantime.

6.6. Environment Querying Interface

The environment querying interface (EQI) is designed with the intent to support the purposes of debugging, testing and statistical evaluation. It is supposed to be concerned with data that is usually not designated to be evaluated by agents, but is rather relevant for the user and/or developer. The EQI reflects tokens of data that are beyond the state of execution and agents-entities-relation, which both have been facilitated by the environment management system and the agents/entities system.

The data that the EQI makes available appears to be allocated as a map, that is key-value pairs where both components are plain Java strings. We distinguish between *entity properties* and *general properties*. Entity properties are considered to be tokens of data that correspond to individual entities and can be queried as such on an individual basis. General properties, on the other hand, are considered to be tokens of data that reflect properties of the environment that are beyond the entities.

Exemplary properties include but are not limited to

1. the overall time spent on executing actions, retrieved for the purpose of profiling the execution of a multi-agent system,
2. the overall number of generated percepts and processed actions, which could be useful for a centralized generation of agent runs,
3. data about the topology of the environment if it is a virtual simulation of reality, and
4. the evaluation of stop conditions for automatic halting of execution.

The EQI is facilitated by two methods:

- `String queryEntityProperty(String entity, String property)` returns an entity's property which is represented by the string `property` and throws an exception if something is wrong, that is if the entity does not exist in the environment.
- `String queryProperty(String property)` yields a general property which, again, is represented by the string `property`.

The use of string as return values of the methods above is no real limitation. Values like floating point and integer numerals or boolean values can be encoded as strings and easily be parsed by the caller of the method. The semantics is highly dependent on the specific environment, which is the reason why we keep the EQI as general as possible.

6.7. Miscellaneous Statements about the Implementation

In order to conclude the elaboration on EIS we now elaborate on two minor details: The versioning system and the partitioning of interface and default implementation.

For the sake of monitoring and maintaining compatibility, EIS requires that each specific interface signals its compatibility with a fixed version of EIS. This is checked

by the interface loading mechanism, which makes sure that the provided environment is compatible to the used runtime of EIS. Each environment has to implement this method:

- `String requiredVersion()` is expected to return a string that denotes the exact version of the EIS runtime that is compatible.

EIS is mainly a contract for agent/platform environment interaction with a couple of programming conventions that the developer is expected to adhere to. We have split EIS into the plain interface contract which is represented by a Java interface and a full-fledged implementation which is an implementation that is faithful to that very contract. The developer now has the options either to make use of the default implementation, which would save him a considerable amount of developing time, or implement his own approach employing the interface, which would require the developer to ensure the conventions himself. Naturally the default implementation can turn out to be quite limited in some aspects, which can easily be alleviated by extending the class, that is overwriting the methods that were considered to be too limited.

6.8. Summary

In this chapter we have elaborated on an interface for environments. The proposed standard was based on an interface meta-model and a set of interface principles that were derived from thorough consideration of the problem of connecting to environments and the previous comparison of APL platforms.

EIS provides an agents-entities system that facilitates the interaction between agents and an environment by relaying actions and percepts. The environment management system, on the other hand, allows to control the state of execution of an environment. The environment querying interface allows for inspecting both the environments entities and its properties that lie beyond entities.

7. Potential Fields and MASs

In this chapter, we elaborate on an application of an important principle of EIS, which we have introduced in the previous chapter: The separation of agents, which provide the intellectual power of a software system, and entities, which establish the agents' situatedness by providing at least sensors and actuators and which are open to provide more properties if required. The main idea in this chapter is to allow a single agent to control a set of entities – that is, virtual vehicles to be more precise – instead of the usual one-on-one relationship.

On top of that, we expect to save a considerable portion of computational power by having one agent control several vehicles. This is made possible by the observation, that vehicles in scenarios similar to the ones we are considering, usually cooperate, form coalitions and thus generally have shared goals. This observation implies that, if we move the shared goals, which closely resemble each other, performance can be improved.

We make use of the *potential fields method* [3, 42, 45, 40] for navigating on a map. This method treats the vehicles in a way that can be compared to particles in an electromagnetic field. By this, we establish a high-level abstraction, since the agent is spared the burden of caring about low-level mechanics of the vehicles. The agent does not have to care about physical principles like position, velocity, acceleration et cetera. This means that low-level actions like turning in a specific direction or moving forward at a given speed are not represented in the agents set of capabilities. Instead, the agent considers the controlled vehicles from a rather high-level perspectives and navigates them on the map by manipulating a potential field that the vehicles share. The agent is capable of putting attractive and repelling forces into the environment. Summing the individual forces up yields a overall force field. The entities are affected by this force field and move accordingly.

Additionally, we have a look at an extension to the potential fields method that adds the well-known A* algorithm to the scenario, that eliminates the major disadvantage of the method. The potential fields method, as it comes, has a problem with complex obstacles, which could lead to vehicles being trapped. We consider and experimentally compare two computer graphics approaches that constitute the potential fields A* combination.

In this chapter, we consider three issues:

- exploring and mapping the environment, that is navigating a set of vehicles on the map in order to generate a mental-state representation of its topology,
- differentiating between accessible and inaccessible areas on the map, that is finding inaccessible areas in the environment and representing those appropriately in the navigation scheme, and finally

- the navigation-scheme itself, that is agent-facilitated steering of vehicles using the potential fields method.

7.1. Motivation, Problem Description and Approach

In the following, we have a (simulated) physical environment in mind. The environment is constituted by a two-dimensional map. Some areas of the map are accessible, while others are not, forming a topology of varying complexity. Situated in that environment is a set of entities, that is robots or vehicles. Complementing the environment, we also have a set of agents that controls these vehicles.

There are several questions that should be answered:

- How can the agents map the environment?
- How can the agents find out which areas are accessible and which ones are not?
- How can agents steer the entities from one place to another?

In our approach, to attack these issues, we employ two types of agents:

1. *entity agents* are associated with the vehicles in the environment, this means that they are associated with respect to the EIS agents-entities-relation, and
2. a *mapper agent*, which is responsible for mapping the environment and providing the entity agents with valid paths and, if requested, with paths to unknown areas in the environment, if exploration is desired.

The mapper agent internally represents the environment in a discretized fashion in the form of a memorized grid. Each cell of the grid is either marked *accessible*, *blocked*, or *unknown*. As already said, this agent is not situated and thus, it cannot act and perceive in the environment. This implies that it has to rely on other agents to provide it with data about the environment.

Each entity agent is situated in the environment, which is facilitated by its associated vehicles. This, of course, means that it is allowed to access the sensors of its associated entities, rendering it capable of perceiving the environment. Furthermore the agent can access the associated entities' actuators, making it possible to act in the environment.

Now, we briefly sketch how the cooperation protocol of the agents looks like. As common in almost all multi-agent systems, all agents are capable of communicating, that is exchanging data via message-passing. Entity agents can request paths from one position on the map to another from the mapper agent. The mapper agent firstly considers its internal environment representation, invokes the A* algorithm to compute a shortest path, and finally communicates the path back to the respective entity agent.

Next is a section on navigating with the potential fields method. There we introduce the main ideas and the mathematical groundwork that facilitates the method.

7.2. Navigating with the Potential Fields Method

The potential fields method solves the problem of environment navigation by representing the topology and structure of the environment as a combination of attracting and repelling forces. In general, a potential field is a function like this:

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

The function maps a two-dimensional[1] coordinate vector to another one. In general the input-vector represents a position on the Euclidean plane. In our case this is a position on the map. The output-vector, however, denotes a force that is effective at that position that usually leads to an acceleration that is effective at the given coordinates. Thus, a moving vehicle is accelerated using the calculated force vector.

As we have already said, force fields are composed by combining several potential fields. We now provide two specialized potential fields, that are quite useful. One is responsible for establishing attraction while the other one is responsible for repellence.

Example 6 (Gaussian repeller) *Any function*

$$f_{gauss} : [x, y] \mapsto \frac{[x - x_0, y - y_0]}{|| [x - x_0, y - y_0] ||} \cdot a \cdot \exp\left(-\frac{|| [x - x_0, y - y_0] ||^2}{2s^2}\right)$$

is called a Gaussian repeller *potential field.*

The constant vector $[x_0, y_0]$ represents the *center*, and the constants a and s represent the *amplitude* and the *spread* of the field respectively. The repelling force is strongest at the center and steeply falls off, converging to 0. In prose, this field is most influential close to its center, while it has almost no effect at a distance. An entity approaching a Gaussian repeller is affected once it gets close to that force. The amplitude a determines the maximum strength of the force. The spread s determines the width of the Gaussian bell and thereby the range of influence of the field.

Example 7 (Sink attractor) *Any function*

$$f_{sink} : [x, y] \mapsto \frac{[x - x_0, y - y_0]}{|| [x - x_0, y - y_0] ||} \cdot g_{sink}(x, y)$$

with

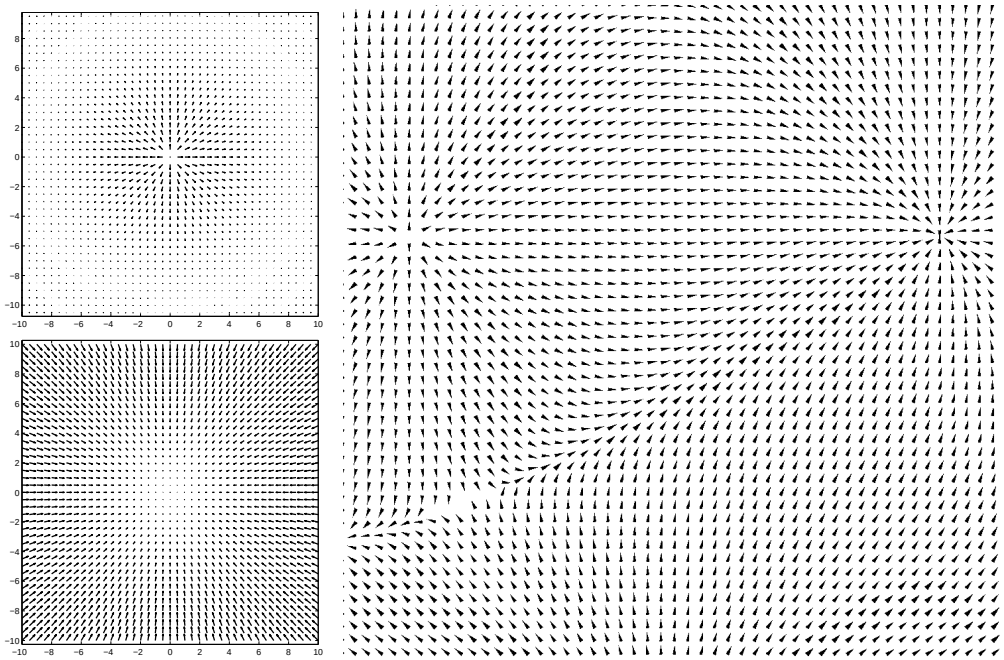
$$g_{sink} : [x, y] \mapsto a \cdot \exp\left(-\frac{|| [x - x_0, y - y_0] ||^2}{2s^2}\right) - g \cdot || [x - x_0, y - y_0] || - a$$

is called a sink attractor.

[1] Of course this approach can easily be extended to more than two dimensions. We remain one the Euclidean plane in this thesis.

The constant vector $[x_0, y_0]$ represents the *center*. The constants a and s represent the *amplitude* and the *spread* respectively. The constant g represents the grade. In layman's terms, the sink attractor's force is stronger the farther away the target is. Mathematically examined, it is a linear combination of a conical potential field and a Gaussian one.

Given a set of potential fields, the overall potential field is straightforwardly determined by calculating the sum of all potential fields that are relevant. Figure 7.1a shows a potential field that is the sum of two Gaussian repellers (top left and bottom) and a sink attractor (top right), all represented as a vector field. A vehicle that starts at a position in the middle would move away from the repellers and move towards the sink.



(a) (b)

Figure 7.1.: Examples for the potential-fields method. The left side shows the Gaussian repeller and the sink attractors. The right side shows the sum of two Gaussian repellers (top left and bottom) and a sink attractor (top right).

The potential fields method is considered to be very elegant, as it is easy to model environmental characteristics with potential fields. Furthermore, it comes handy when dealing with several moving entities: This is why we have chosen this approach. An AI programmer using that method does not have to deal with movement on a microscopic level for each vehicle. Instead she just lets them follow a shared potential field, and they would flow like particles in a stream.

Although the method is elegant and allows vehicle movement in a unified way, there are also drawbacks. The worst drawback is the fact that vehicles can (and often will)

get stuck if there are local optima in the potential field. A possible solution would be to move a stuck entity a bit, for example to an arbitrary direction. This could release the vehicle from the local optimum, if the overall potential field is not very complex. But if the opposite is the case this method is rendered useless. Path-planning, however, as we show soon, does improve the situation significantly.

7.3. Path Based Potential Fields

In order to reduce the probability of a vehicle getting stuck in a local optimum, we come up with a compromise approach. We employ the well-known A* algorithm and use its outcome to generate specialized potential fields, which we call *path based potential fields*. In our approach, we distinguish two classes of obstacles: *static* and *dynamic* ones. We avoid static obstacles by path-planning, and dynamic ones through the considerate placement of repellers, both in an unified framework.

The A* algorithm is an informed search algorithm for finding shortest paths in graphs using a heuristics [55]. In our approach, we use an implicit representation of the search graph, which is a grid model of the environment. We define each cell of such a grid that is reachable as a node of the graph. Figure 7.2 shows a straightforward visualization of that situation. The set of nodes of the graph is equivalent to the subset of reachable cells of the map. Edges in the graph are defined as follows. Two nodes are adjacent if the respective cells share a vertex or an edge. This implies that horizontal, vertical and diagonal movements are possible. The *costs* of traversing the edge is either 1 if the respective cells are horizontal/vertical neighbors, and $\sqrt{2}$ if they are diagonal ones. The *heuristics*, on the other hand, is the distance as the crow flies, making the assumption that we calculate the distance between the centers of the respective cells, while we also assume that the cells are unit-squares.

We now introduce, consider and compare two computer graphics algorithms for generating specialized potential fields from paths yielded by the A* algorithm using the graph representation of the environment we have elaborated on above. The first algorithm is geometry-based, that is it deals with polygons, while the second one is raster-graphics based, which means that it deals with bitmaps of appropriate sizes and resolutions. We ask and answer the following question: given a path $P := (p_0, p_1, \ldots, p_n)$ with waypoints $p_i := [x_i, y_i] \in \mathbb{R}^2$, which has been generated by applying the A* algorithm on the map graph, how can a specialized potential field be generated that makes the vehicles follow the path in a smooth manner?

7.3.1. Geometry Based Approach

The geometry based approach operates as follows. A given path is employed to generate a strip of polygons, while introducing one polygon for two consecutive points of the path. Each polygon, which is a quad, that is a polygon with four points, is then prepared to generate a force field. Calculating a force that is effective at a given position is then reduced to finding out how the given position is positioned in relation to the individual quads.

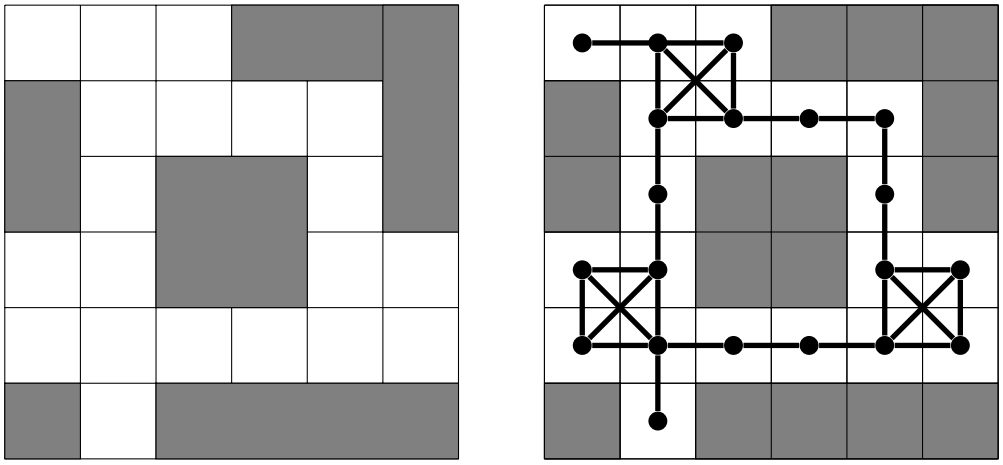


Figure 7.2.: These images display the discretization of the environment's topology that we use throughout the chapter. The left image shows the map as a grid that differentiates between accessible (white) and inaccessible (black) areas. The right image shows the search graph for the A* algorithm.

The approach works in several phases. In the *preprocessing phase*, given a path

$$P := (p_0, p_1, \ldots, p_n)\,,$$

we generate a *force-quad strip*, which consists of one *force-quad* for each path segment $\langle p_i, p_{i+1} \rangle$. A force-quad is a tuple

$$q := \langle ls, le, ll, lr, d \rangle\,,$$

where $ls, le, ll, lr \subset \mathbb{R}^2$ are the lines defining the shape of the quad, and $d \in \mathbb{R}^2$ is the direction of its force. A force-quad strip is a sequence

$$Q := (q_0, q_1, \ldots, q_m)$$

with $m = n - 1$ and where for each pair q_i, q_{i+1} the following holds: $le_i = ls_{i+1}$. Figure 7.3a shows an exemplary segment of a force-quad strip.

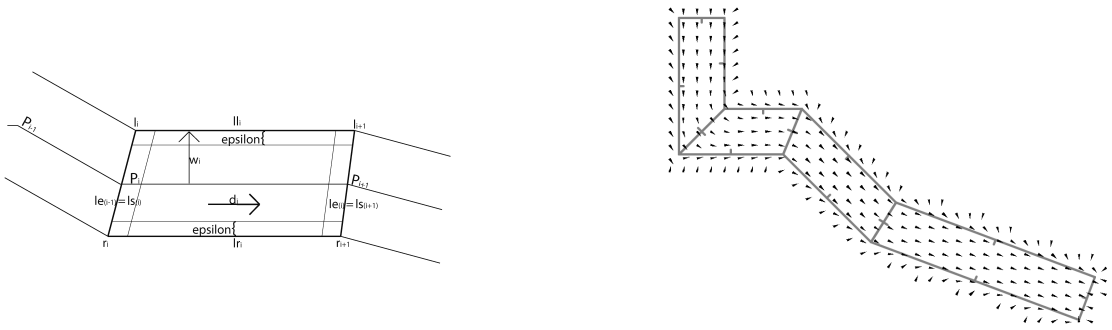
Calculating the force vector $f \in \mathbb{R}^2$ at a given environment position $p \in \mathbb{R}^2$ with respect to a given force-quad strip Q boils down to 1. detecting which force-quad

$$q_i := \langle ls_i, le_i, ll_i, lr_i, d_i \rangle \in Q$$

contains p and 2. calculating f depending on d_i.

Preprocessing

The input-parameters for the force-quad strip generation are 1. the path P as defined above, and 2. a width $w \in \mathbb{R}^+$, which is assumed to define how broad the effective



(a) (b)

Figure 7.3.: Illustrations for the force-quads based approach. The left side shows the lines, vectors, and values that the preprocessing algorithm makes use of. The right side shows a plot of the force field.

force field is supposed to be. Algorithm 1 creates a sequence of connected force-quads, whereas all quads have the same width and their ordering reflects the ordering of the waypoints of the path. Firstly, the direction vector $d_i := p_{i+1} - p_i$ is calculated for each path segment $\langle p_i, p_{i+1} \rangle$. The vector w_i is orthogonal to d_i and has the length w. The line ll_i is $\langle p_i, p_{i+1} \rangle$ translated by w_i and the line lr_i is $\langle p_i, p_{i+1} \rangle$ translated by $-w_i$. The algorithm then calculates the intersections of ll_i with ll_{i-1} and ll_{i+1} respectively, and the intersections of lr_i with lr_{i-1} and lr_{i+1} respectively. These four intersections define the force-quad. This only works for those points, that have both a predecessor and a successor. The first and the last quad, on the other hand, are both special cases:

1. for the first quad the lines l_{-1} and r_{-1} are both set to $p_0 + -w_0 - d_0$, and
2. for the last quad the lines l_n and r_n are both set to $p_n + -w_n - d_n$.

This distinction of cases allows handling the special case that the path consists of only two waypoints. Figure 7.3a shows a quad within a given path as well as all lines and vectors that the algorithm uses.

Calculating Force Vectors

After generating a representation of the specialized path-based potential field in the form of a strip of quads, we now consider the algorithm that computes the force-vector f for a given position $p \in \mathbb{R}^2$ with respect to a force-quad strip Q. The algorithm iterates over all the force-quads $q_i \in Q$ and checks if p is contained by q_i. If so, the algorithm computes the force-vector with respect to that quad, using its direction vector. If p is not contained by any of the quads, the null vector is returned.

In order to get a smoother movement of the vehicles, the algorithm interpolates d_i of the force-quad q_i that contains p with respect to the position of p in q_i. This means that the algorithm determines if the distance $dist$ of p to one of the borders of the quad is smaller than a given ϵ. If that is the case, the algorithm uses the linear interpolation of

Algorithm 1 Preprocessing: generating a force-quad strip

Require: a path $P := (p_0, p_1, \ldots, p_n)$ and a width $w \in \mathbb{R}^+$
Ensure: a force-quad strip $Q := (q_0, q_1, \ldots, q_m)$ with $m := n - 1$
  $Q := \emptyset$;
  for all $i \in [0, n-1]$ **do**
    $d_i = Normalize(p_i - p_{i+1})$
    ll_i is (p_i, p_{i+1} translated by w_i; lr_i is (p_i, p_{i+1} translated by $-w_i$;
  end for
  if only two waypoints **then**
    for all $i \in [0, n-1]$ **do**
      if $i = 0$ **then**
        $l_i = p_0 + w_i - d_i$; $r_i = p_0 - w_i - d_i$
      else
        $l_i = Intersection(ll_{i-1}, ll_i)$; $r_i = Intersection(lr_{i-1}, lr_i)$
      end if
      if $i = n - 1$ **then**
        $l_{i+1} = p_n + w_i - d_i$; $r_{i+1} = p_n - w_i - d_i$
      else
        $l_{i+1} = Intersection(ll_i, ll_{i+1})$; $r_{i+1} = Intersection(lr_i, lr_{i+1})$
      end if
      $ls_i = line(l_i, r_i); le_i = line(l_{i+1}, r_{i+1})$
      Save Quad $\langle ls_i, le_i, ll_i, lr_i, d_i \rangle$ in Q
    end for
  else
    $l_0 = Intersection(ll_0, (p_0 - d_0 + w_0))r_0 = Intersection(lr_0, (p_0 - d_0 - w_0))$
    $l_1 = Intersection(ll_0, (p_1 + d_0 + w_0))r_1 = Intersection(lr_0, (p_1 + d_0 - w_0))$
    $ls_0 = line(l_0, r_0); le_0 = line(l_1, r_1)$
    Save Quad $\langle ls_0, le_0, ll_0, lr_0, d_0 \rangle$ in Q
  end if
  return Q

d and the normal vector n to calculate the resulting vector f. This is described by the following formula:

$$f = (dist/\epsilon) * d + (1 - (dist/\epsilon)) * n$$

Algorithm 2 computes the force-vector. And Figure 7.3b shows a vector-field generated using the computed force-quad strip, by laying a grid of input vectors over the force field and determining the force that is effective.

7.3.2. Image Processing Approach

After the geometry based approach, in this section, we describe a method that represents a specialized potential field in a discretized fashion. To that end, we derive a bitmap

Algorithm 2 Calculating Force Vectors

Require: a position vector $p \in \mathbb{R}^2$ and a force-quad strip $Q := (q_0, q_1, \ldots, q_m)$ threshold $\epsilon \in \mathcal{R}^+$
Ensure: a force vector $f \in \mathbb{R}^2$
  for all $q \in Q$ **do**
    if p is contained by q **then**
      $f := interpolate(p, q, \epsilon)$
      return f
    end if
  end for
  return $[0, 0]$

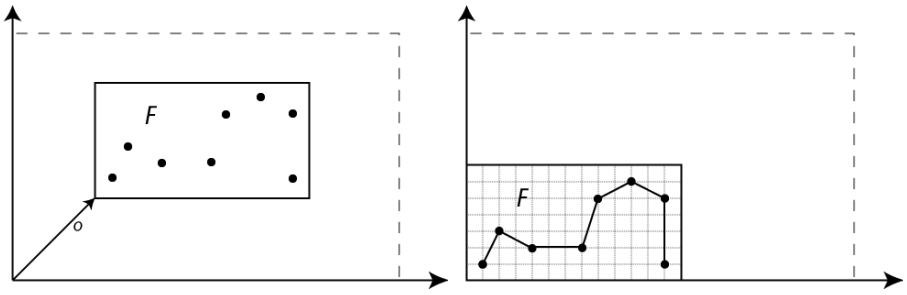


Figure 7.4.: The local frame F. The left side shows its alignment in the environment and the waypoints. The right side shows F after translation by o and plotting the path.

from a given path $P := (p_0, p_1, \ldots, p_n)$, while borrowing some machinery from image processing.

The idea is to take a given path and use it to generate a height map, which contains the path as a gradient. Movement would than be reduced to performing a gradient descent towards low areas. Again, we introduce several methods. During the preprocessing phase, we generate the aforementioned height map. To calculate force vectors we perform a lookup on the height map and derive directions from neighboring pixels.

Preprocessing

Firstly, we compute a bounding box that contains the whole path. The bounding box's dimensions are used to initialize a height map, which is afterwards drawn upon. The bounding box is then slightly enlarged to create a local frame, which is the rectangle on the map in which the considered path is expected to be effective. We define the local frame as $F \subset \mathbb{R}^2$. To facilitate later transformations, we translate a single corners of F into the origin of the coordinate system by subtracting the vector $o \in \mathbb{R}^2$, as shown in Figure 7.4. The derived transformation is finally applied to the individual waypoints of the path.

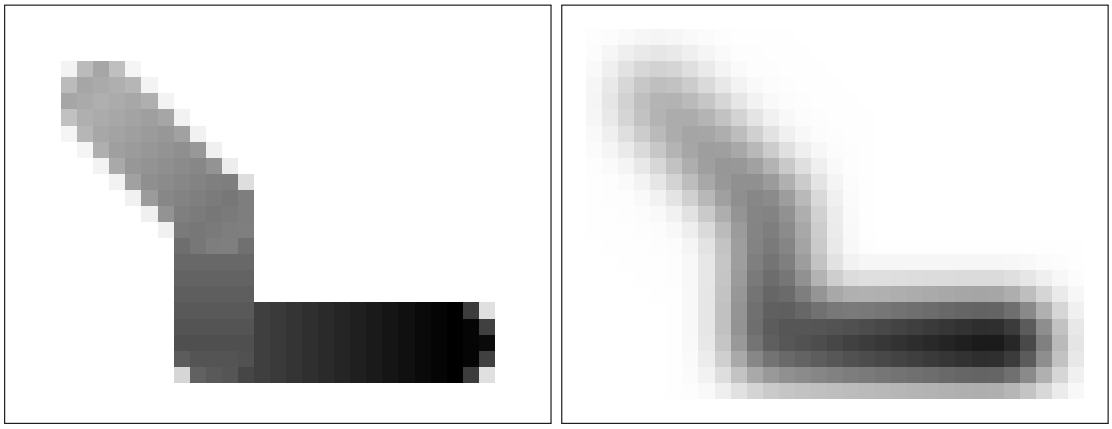


Figure 7.5.: The same path after applying two Gaussian operations with different intensities. The higher the intensity the smoother the movement.

We introduce some factors $r_x, r_y \in \mathbb{Z}$, which are supposed to facilitate an appropriate resolution for the resulting bitmap. It should be noted, that the parameters can be defined by the mapping agent. The axes of the coordinate system are scaled by these factors, resulting in a discrete representation of the local frame F, which we from now on consider to be a bitmap. After allocation, the image is filled with an initial height (color) value. Then for each pair of consecutive waypoints we invoke a discrete line drawing algorithm with a given line-width w. This width, however, depends on the use case. A thinly rendered line generates a single file movement for the vehicles, whereas a thick line would allow the vehicles to maintain their formation. The line strip is rendered with a color gradient starting with a highest and ending with a low intensity value, as shown in Figure 7.5. For reasons of optimization the resulting bitmap is then slightly blurred in order to ensure smooth movement. This is done by applying a Gaussian operator.

Algorithm 3 Preprocessing

Require: a path $P := (p_0, p_1, \ldots, p_n)$ a scaling factor r, the line-width w, and the blur intensity b

Ensure: $< F, T >$ the local frame F and the transformation T

$B := calcBoundingBox(P)$
$F := initImage(B.width/r, B.height, /r, highestIntensity)$
$T := getTransformation(B, r)$ {maps from environment to frame coordinates}
for all p_i **do**
  $p_i := T(p_i)$ {map all waypoints to frame-coordinates}
end for
$F := drawPolyLine(F, P, highIntensity, lowIntensity, w/r)$
$F := blur(F, b)$
return $< F, T >$

Calculating Force Vectors

Now, after the bitmap containing a height map for a gradient descent has been computed, we have a look at how force vectors can be calculated using this very bitmap.

We assume that query point $q \in \mathbb{R}^2$, that is an arbitrary position on the map, is given. Our algorithm performs these three steps to retrieve the respective force vector:

1. transforming the environment position p into the respective position p' in the frame by translating and scaling p respectively, using the transformation that has been computed in the preprocessing phase,

2. calculating the intensity difference between the pixel p' and all of its eight direct neighbors, and

3. retrieving the force with respect to the difference of intensities.

The pseudo code for the retrieval is given in Algorithm 4.

Algorithm 4 Calculating Force Vectors

Require: a position vector $p \in \mathbb{R}^2$, a local frame F and a transformation T
Ensure: a force vector $f \in \mathbb{R}^2$

```
q := T(q) {map to frame coordinates}
c_q := getColorAt(F, q)
prevDiff := 0; m := 0; n := 0
for i := −1 to 1 do
  for j := −1 to 1 do
    diff := c_q − F(q[x] + i, q[y] + j)
    if diff > prevDiff then
      m = i; n = j; prevDiff = diff
    end if
  end for
end for
return (m, n) ∗ prevDiff
```

7.4. Implementation and Comparison

Up to this point, we have introduced the potential fields method and proposed a partial solution to the local optima problem that arises when applying the method. We have suggested an extension to the method that makes use of specialized, path-based potential fields. To generate such force fields we have designed and elaborated on two approaches. The first one is based on a geometrical technique that represents the potential field as a sequence of polygons, and allows to derive effective forces at given coordinates in the polygons. The second one is bitmap based. It represents the potential field as a height map and allows to calculate effective forces based on a gradient descent idea.

Now, we intend to compare the two approaches using appropriate experiments. The basis of our experiments is a simulated physical world, reminiscent of common computer-game like scenarios. As a side note it has to be said, that a lot of computer games make the cooperation of entities a desideratum, especially the cooperation between virtual entities and the human player. Usually computer games provide a highly detailed simulation. This holds in particular, when it comes to the physical model and the aural/visual representation of the game events. Of course, we are only interested in agents interacting with the physical simulation, while neglecting the graphics and acoustics. For many, but definitely not for all computer games, it is straightforward to either rely on already existing sensors and actuators or create new ones for agents that are supposed to be situated in a game world.

Nevertheless, we base our experiments on a simplified and custom-made environment. Our experiences and results, of course, could have been mapped to a real computer game, but we decided against this. The environment is a plane with obstacles. On that plane is a set of vehicles that can move in any direction if not blocked by an obstacle, and that can perceive the plane in their vicinities.

For the comparison of the two approaches[2], we have set up a very intuitive scenario. Our starting point is a path with 21 nodes as depicted in Figure 7.6. In each run of our experiments we generate two potential fields, one is geometry based and the other is bitmap based, and randomly query for force vectors for a fixed number of times, that is one million times. After that we remove the last point of the path and repeat our measurements until we end up with a two-points path, which concludes the experiments. In each step we measure the time that is consumed for each of the two approaches.

# Nodes	s Quads	s Raster
21	5.9	5.8
19	5.6	5.8
17	5.5	5.8
15	5.1	5.8
13	4.6	5.8
11	4.2	5.7
9	3.9	5.8
7	3.1	5.8
5	2.3	5.6
3	2.4	5.4

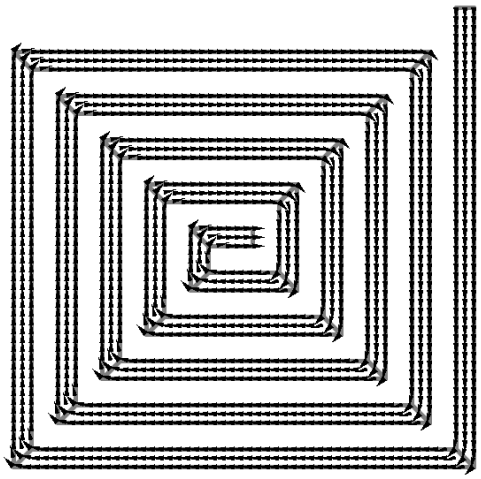


Figure 7.6.: Comparison of the two approaches. The left side show the results. The right side shows the path that has been used.

[2]The experiments were done on a 2 GHz Intel Core Duo MacBook Pro with 2GB RAM.

The table in Figure 7.6 clearly shows that the geometry/polygon based algorithm performs worst for long path. This actually has been expected, because we saw that for each added node, we have to compute four additional dot-products. The image based algorithm on the other hand always consumes the same amount of time. Viewing from an implementation perspective, implementing the preprocessing step of the bitmap-based approach is easier, because it can be reduced to using a graphics API, which is usually readily available. It turned out that for short path the bitmap-based approach is more memory consuming. The polygon-based algorithm requires the space of $(n-1) \cdot 14$ doubles, when a path-length of n is given. The bitmap algorithms consumption of memory depends on the area and resolution of the area covered by the path. From an aesthetics stance, we can say that the polygon-based approach yields a slightly more convincing movement of entities.

7.5. Summary and Related Work

In this chapter, we began with the application of EIS's agents-entities relation that allows a single agent to control several entities in the environment. We have then introduced the potential fields method which relieves an agent of the burden to control his associated entities, in our consideration simulated vehicles, on a microscopic level. Instead the potential fields method allows the agent to steer a set of entities by manipulating force fields. To go even deeper, we then elaborated on specialized potential fields and compared two approaches.

In the area of potential fields Hagelbäck et al. [33, 32] did similar research. They use a discretization for the complete map, which represents a force-field for the entire topology. When it comes to the structure of the used multi-agent systems, our approaches differs significantly. Their idea is to have each vehicle controlled by one *unit agent*, and another agent that coordinates those unit agents. We do not consider entities to be agents. In their paper [41], Koren and Borenstein discuss problems inherent to the potential fields method. They identify four significant problems: trap situations due to local optima, no passage between closely spaced obstacles, oscillations in the presence of obstacles, and oscillation in narrow passages. Mamei and Zambonelli [44] apply potential fields to the computer game Quake 3 Arena.

8. EISMASSim for the Multi-Agent Programming Contest

In this chapter, we merge two of our research lines. We consider a combination of EIS, the environment interface standard for agent-oriented programming, and *MASSim*, the software architecture that facilitates the Multi-Agent Programming Contest. We concentrate on *EISMASSim*, which is an environment interface for the agents on Mars scenario. This shows how EIS can be used in combination with a greater and established project. Additionally, we show that *EISMASSim* simplifies working with *MASSim* and is faithful to EIS's reusability principle. The environment interface makes the communication with the server easier, because it reduces the XML-based communication protocol to method calls and callbacks. Altogether, we make clear how an environment can be EISified, connecting the environment and the environment interface via TCP/IP.

The Multi-Agent Programming Contest[1] was conceived and established in 2005 and since then went through four phases, that is

1. the *food gatherers* phase in 2005 in which participants were given the task of implementing a multi-agent system, that is a set of agents and an environment, for a scenario of agents collecting food in a grid-world,

2. the *gold miners* phase from 2006 to 2007 in which participants were given the task of implementing a set of agents for a scenario provided by the organizers, in which agents should collect gold while outperforming other teams,

3. the *cows and cowboys* phase from 2008 to 2010 in which sets of agents should be implemented that hunt and gather virtual cows while outperforming other sets of agents, and

4. the *agents on Mars* phase in which sets of agents compete on planet Mars for resources.

In this thesis, we do not concentrate on the very first phase, because it is the *MASSim* software, which is relevant and which has facilitated the Contest since 2006. In this chapter, we elaborate on an EIS-compatible connector for *MASSim*, that we call *EISMASSim*. *MASSim* (see Figure 8.1) is the software-backbone of the Contest and consists of but is not limited to these components:

- an infrastructure for agent-environment interaction, that is based on TCP/IP and XML and that allows agents to perform actions in the environment and perceive its state,

[1] `http://multiagentcontest.org`

- a scenario executor that loads and executes a scenario plug-in, that is the actual environment,
- a tournament scheduler that executes the overall tournament,
- an interface to web-based applications that allows for monitoring the tournament and the simulations, and
- a video generator that renders each simulation for later examination.

In its totality, *MASSim* is capable of organizing a tournament consisting of a multitude of simulations between a set of different participating teams. Usually each team plays against all other teams, each time for a fixed number of simulations. Again, because we are only interested in the environment aspect of the Multi-Agent Programming Contest, that is the agents-environment interaction, we focus our attention only on this aspect while ignoring the tournament-aspect for the time being.

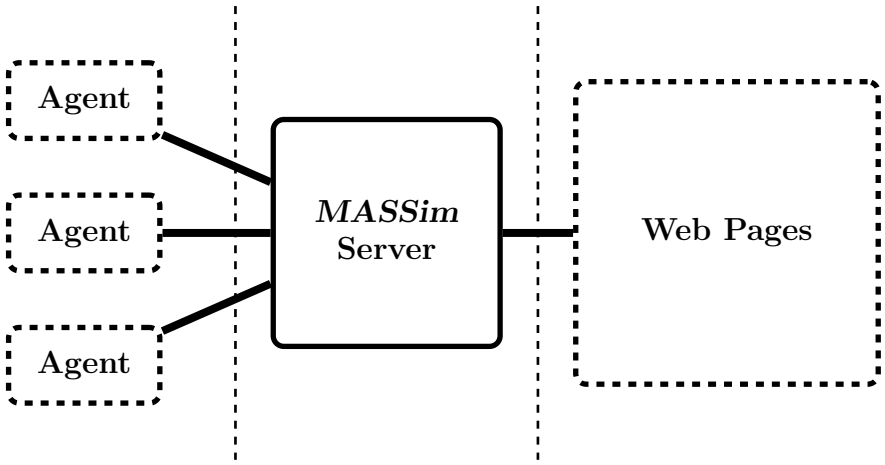


Figure 8.1.: This is a schematic view of the Contest set-up. The *MASSim* server is the center of the arrangement. It schedules and executes the simulations that constitute the tournament. Agents are clients on the participants' machines. The overall tournament and the single simulations can be monitored via web clients.

All scenarios used throughout the history of the Contest can be characterized by the following properties: they are 1. *inaccessible*, that is each agent did not perceive the respective environment as a whole and only had a limited view of the environments' state, 2. *nondeterministic* , that is actions can fail with a specific probability with usually some randomness being involved in the evolution of the environment, 3. *non-episodic*, that is the states do not evolve episodically, 4. *dynamic*, that is the state of the environments can change without the agents acting (gold can appear randomly, and cows move automatically), and 5. *discrete*, that there is a limited (but huge) set of environment states and the environment evolves in steps.

In the next sections, we firstly elaborate on a generalized EIS-compatible environment interface for *MASSim* called *EISMASSim*. After that, we deal with the features of *EISMASSim* that are specific to the different scenarios. In addition, we briefly discuss the different scenarios, and show how perceiving and acting is facilitated by the environment interface. Finally, we elaborate on the mechanism that allow for querying *EISMASSim* in order to retrieve statistics.

8.1. General Environment Interface

In this section, we elaborate on the general environment interface for the Multi-Agent Programming Contest, that we call *EISMASSim*. This general environment interface is supposed to establish the connection to any *MASSim*-server and execute the communication-protocol of all scenarios. We identify the following requirements for *EISMASSim*:

- *arbitrary number of connections*, that is the user can instantiate an arbitrary number of connections to a *MASSim*-server located somewhere on the internet and authenticate each such connection via username and password,
- *connection stability*, that is the environment interface is expected to ensure that if a connection is lost during runtime it is reestablished,
- *translating the communication*, that is the environment interface retrieves XML documents from the *MASSim*-server and maps them to the IILang, while mapping actions notated in the IILang are mapped to valid XML documents and sent to the *MASSim*-server,
- *support for all scenarios*, that is a single environment interface is available that handles the connection to all scenarios, which is feasible because – from a technical point of view – they are equivalent when it comes to the agent-environment interaction [2],
- *logging facilities*, that is displaying and storing for later retrieval the agent view of the agent-environment interaction, that is the received percepts and performed actions, for debugging and monitoring purposes, and
- an *interface for queryable properties and statistics*, that is the environment interface should provide means for inspecting the agent-environment connection and gathering statistics about it.

The *EISMASSim* environment interface consists of the following key components:

- A *set of connectors* facilitates the communication with *MASSim* via TCP/IP. This component is a point for customization. For each scenario a new connector must be defined.

[2] In this thesis we only focus on the newest scenario.

```
<?xml version="1.0" encoding="UTF-8"?>
<interfaceConfig scenario="mars2011" host="ag1.multiagentcontest.org"
port="12300">
  <entities>
    <entity name="vehicle1" username="teamA1" password="topSecret"/>
    <entity name="vehicle1" username="teamA2" password="topSecret"/>
    ...
  </entities>
  <logger toTerminal="yes" toFile="yes" toFiles="yes"/>
  <backup file="backup1.xml"/>
</interfaceConfig>
```

Figure 8.2.: An exemplary XML configuration file for *EISMASSim*.

- The *scheduler* is an optional component which ensures that each connector synchronizes itself with the server by intelligently blocking access to actuators and sensors.

- The *inspection module* allows access to data held available for the purpose of creating statistics.

- The *configuration module* provides means to customize the instantiation of the environment interface.

An instance of *EISMASSim* is set up by providing a config-file written in XML. In this file it can be specified which scenario should be used. Possible scenarios are `goldminers2006`, `goldminers2007`, `cowboys2008`, `cowboys2009`, `cowboys2010` and finally `mars2011`. Agent-connections with *MASSim* require from the agent-developer to create a TCP/IP socket and then authenticate her agents using the credentials, that is a username and a password for each agent. These credentials are stored in the XML-file. This is the right level to do so, because agents are not supposed to be aware about the details of the connection. Finally, it can be specified how the logger works, that is whether it should log everything to the shell, log everything in a single-file, or log everything in a single file for each agent. Figure 8.2 shows an exemplary configuration file.

A connector is a class that is responsible for the execution of two important tasks. XML-messages that were sent by the *MASSim*-server are parsed and split into individual percepts. To enable communication in the opposite direction, actions are transformed into XML-messages and sent to the server. Because the different incarnations of the agent contest, that is the individual scenarios, differ significantly in the contents of the action- and percept-messages, a specialized connector has to be created for each scenario, that implements the required generating- and parsing-functionality.

After the instantiation phase, *EISMASSim* firstly attempts to establish an individual TCP/IP connection for each connector specified in the configuration. Once this attempt is successful, *EISMASSim* works off the three phases of the client-server communication protocol, again, for each connector. In the first phase, a valid authentication message, containing a valid username and password, is composed and sent to the server. This message is expected to be acknowledged by the server, which sends a respective message in return. Establishing a valid connection to the *MASSim*-server means 1. creating a socket to the server specified in the configuration file, that is its host-name and port-number, 2. sending an XML-message with the type `AUTH-REQUEST` that contains the username and the password, and 3. waiting for a notification that the authentication has been successful in the form of an `AUTH-RESPONSE`-message.

Following that, in the second phase, the sense-act part of the communication protocol is handled. In every step the *MASSim*-server issues a message containing percepts and then expects a reply containing an action. We distinguish three types of percepts: 1. percepts at the simulation *start*, 2. percepts *during* the simulation and 3. percepts at the simulation *end*. At the beginning of each simulation, the agents are provided with *static* information, that is information which does not change during the simulation. *Dynamic* information, on the other hand, is provided at each step of the simulation and contains the local view of the agent. The provision of percepts during the simulation is also the request for the agent to act. At each step, the agent is allowed to execute one action. At the end, the agent receives information about the outcome of the simulation.

Finally, in the third phase, the *MASSim* server informs the agents about the end of the tournament and informs that the TCP/IP connections can be terminated.

Because *MASSim*'s means of communication are XML-based, each of these examined tokens of communication have to be mapped from and to IILang. Since the syntax of the XML-messages is highly specific to the respective scenario, we elaborate on these mappings when we examine the scenarios in detail. At this point of the thesis, we can elaborate on the messages, that are not scenario-specific, that is establishing an authenticated connection which is acknowledged by the server and notifying about the end of the tournament.

8.2. Agents on Mars

In the *Agents on Mars*-scenario, teams of robots are situated on planet Mars with the goal to find and conquer valuable water-wells, while competing with other teams that have the same goal. Tasks are 1. to explore the environment, 2. to organize the robots that have different roles and 3. to get achievements. The overall demand to an agent team is to optimize both the value of conquered zones and the points derived from achievements.

The environment is constituted by a static graph, as shown in Figure 8.3, consisting of vertices V and edges E. Each vertex has a value, represented by value-function $value : V \to \mathbb{N}$ for the whole graph. Analogously, each edge has a weight, denoted by the weight-function $weight : E \to \mathbb{N}$. The value of the vertices is crucial for the scoring

mechanism. Groups of robots can cover a set of vertices, which we call a *zone*. The value of such a zone is the sum of the values of covered nodes. The edge-values, on the other, hand denote the costs of traversing the edges, which is relevant when it comes to cruising range and speed of the robots.

The robots are the entities in the EIS-sense. They have different roles, and thus different capabilities. Each entity has the following properties:

- position on the graph that is the vertex it is standing on,
- an identifier representing its unique name,
- another identifier denoting its team's name,
- an identifier that represents the entity's role,
- a numeral that depicts the energy,
- another numeral that represents the health, and
- a final numeral that represents the visibility-range.

There is no restriction on how many entities are on one and the same node. The energy is usually reduced by a fixed number when an action is executed. Health, however, is reduced when an attack occurs and is successful.

The entity's capabilities are as follows. Each robot can

1. move to an adjacent vertex in the graph,
2. probe the vertex the entity is standing on to determine its value,
3. survey the adjacent edges to determine their weights,
4. inspect the visible opponent entities,
5. attack an entity that is standing on the same node,
6. parry an expected attack,
7. buy upgrades,
8. repair another entity, and
9. recharge its energy.

Each entity can assume one of the following roles. The assignment is predefined in the server-configuration file and does not change during the simulation/tournament. Most significantly the roles differ when it comes to the initial values for energy, health and visibility-range. An entity can either be

1. an *explorer* that can move, probe, survey, buy and recharge,

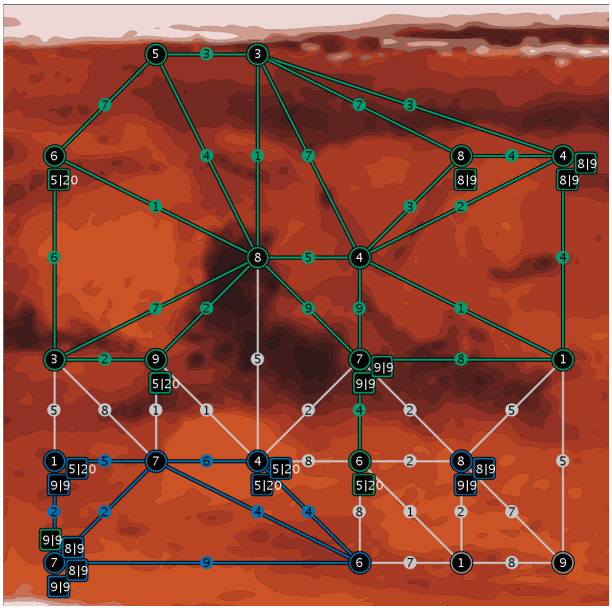


Figure 8.3.: A screenshot of the Agents on Mars scenario. The agents control a team of robots that are on a grip representing a fictional Mars landscape. The robots can perceive objects in its visibility range and can interact with the environment.

2. a *repairer* that can move parry, survey, buy and recharge,
3. a *saboteur* that can move, parry, survey, buy, attack and recharge,
4. a *sentinel* that can move, parry, survey, buy and recharge, and
5. an *inspector* that can move, inspect, survey, buy and recharge.

The evolution of the environment is facilitated by the following steps:

1. sending percepts to the agents,
2. waiting for actions or performing a time out,
3. letting received actions fail with a fixed probability,
4. executing remaining move actions, and
5. executing remaining attack-defense actions.

8.3. Actions and Percepts for the Mars-Scenario

In the following, we elaborate on actions and percepts expressed in the IILang. Each action and each percept consists of a name followed by an optional list of parameters. A parameter is either an identifier (`<Identifier>`), that is a string, or a numeral(`<Numeral>`).

Here is the list of actions that can be performed in the course of each simulation[3]:

- `attack(<Identifier>)` attacks another robot,
- `buy(<Identifier>)` buys an item,
- `goto(<Identifier>)` moves to a vertex,
- `inspect` inspects some visible vehicles,
- `parry` parries all imminent attacks,
- `probe` probes the current vertex,
- `recharge` recharges the vehicle,
- `repair(<Identifier>)` repairs a vehicle,
- `skip` does nothing, and
- `survey` surveys some visible edges.

In the following, we consider a list of percepts that can be available during a tournament. Note that during a simulation, data from the respective `sim-start`-message is available as well as data from the current `request-action`-message (see the protocol description for details about such messages):

- `achievement(<Identifier>)` denotes an achievement,
- `bye` indicates that the tournament is over,
- `deadline(<Numeral>)` indicates the deadline for sending a valid action-message to the server in Unix-time,
- `edges(<Numeral>)` represents the number of edges of the current simulation,
- `energy(<Numeral>)` denotes the current amount of energy of the vehicle,
- `health(<Numeral>)` indicates the current health of the robot,
- `id(<Identifier>)` indicates the identifier of the current simulation,
- `lastAction(<Identifier>)` indicates the last action that was sent to the server,
- `lastActionResult(<Identifier>)` indicates the outcome of the last action,

[3]See the scenario's description for the precise semantics of the actions.

- `lastStepScore(<Numeral>)` indicates the score of the vehicle's team in the last step of the current simulation,
- `maxEnergy(<Numeral>)` denotes the maximum amount of energy the vehicle can have,
- `maxEnergyDisabled(<Numeral>)` denotes the maximum amount of energy the vehicle can have, when it is disabled,
- `maxHealth(<Numeral>)` represents the maximum health the vehicle can have,
- `money(<Numeral>)` denotes the amount of money available to the vehicle's team,
- `position(<Identifier>)` indicates the current position of the vehicle. The identifier is the vertex's name,
- `probedVertex(<Identifier>,<Numeral>)` denotes the value of a probed vertex. The identifier is the vertex's name and the numeral is its value,
- `ranking(<Numeral>)` indicates the outcome of the simulation for the vehicle's team, that is its ranking,
- `requestAction` indicates that the server has requested the vehicle to perform an action,
- `score(<Numeral>)` represents the overall score of the vehicle's team,
- `simEnd` indicates that the server has notified the vehicle about the end of a simulation,
- `simStart` indicates that the server has notified the vehicle about the start of a simulation,
- `step(<Numeral>)` represents the current step of the current simulation,
- `steps(<Numeral>)` represents the overall number of steps of the current simulation,
- `strength(<Numeral>)` represents the current strength of the vehicle,
- `surveyedEdge(<Identifier>,<Identifier>,<Numeral>)` indicates the weight of a surveyed edge. The identifiers represent the adjacent vertices and the numeral denotes the weight of the edge,
- `timestamp(<Numeral>)` represents the moment in time, when the last message was sent by the server, again in Unix-time,
- `vertices(<Numeral>)` represents the number of vertices of the current simulation,
- `visRange(<Numeral>)` denotes the current visibility-range of the vehicle,

- `visibleEdge(<Identifier>,<Identifier>)` represents a visible edge, denoted by its two adjacent vertices,
- `visibleEntity(<Identifier>,<Identifier>,<Identifier>,<Identifier>)` denotes a visible vehicle. The first identifier represents the vehicle's name, the second one the vertex it is standing on, the third its team and the fourth and final one indicates whether the entity is disabled or not,
- `visibleVertex(<Identifier>,<Identifier>)` denotes a visible vertex, represented by its name and the team that occupies it,
- `zoneScore(<Numeral>)` indicates the current score yielded by the zone the vehicle is part of, and
- `zonesScore(<Numeral>)` indicates the current score of the robot team yielded by zones, that is the sum of scores of all zones.

8.4. Statistics

In 2011, we automatically collected data from simulations in the later organizational phase of the contest. To that end, we have facilitated a statistics mechanism for relaying significant data to our servers. Such data is gathered by a statistics core class for every instance of *EISMASSim* and submitted via a straightforward web-API that is constituted by PHP and HTTP web requests. That is, while a tournament is running, *EISMASSim* gathers data and submits the collection as a whole once the tournament is over. This mechanism has been designed in order to give us some statistics that helped us to balance the scenario parameters. We gather such data:

- *average time between request and response*, that is the average time between an agent receiving a set of percepts and sending an action to the server,
- *per action data*, that is for each type of action its overall percentage with respect to the collection of all performed actions,
- *per percept data*, that is for each type of percepts its total number of percepts received,
- *per agent data*, that is for each agent its average response time, the values of the zones it helps to occupy, the number of percepts sorted by type, the total number of performed actions, the total number of failed actions, a percentage overview of all the agent's actions, and the total percentage, and
- *per team data*, that is the overall zone-scores and the achievements.

8.5. Summary

In this chapter, we have bridged the gap between EIS, a proposed standard for agents-environment interaction, and *MASSim*, the software infrastructure that is the groundwork of the Multi-Agent Programming Contest. We have shown an environment interface, named *EISMASSim*, which facilitates a connection between agents and an environment via TCP/IP. On one side, *EISMASSim* communicates with *MASSim* by exchanging messages. On the other side *EISMASSim* communicates with agents via method invocations and callbacks.

9. APLTK: A Toolkit for Agent-Oriented Programming

In the previous chapter about EIS, we have focussed on connecting agents and agent platforms to external environments. In this chapter, we continue after this first step towards the overall goal of this thesis and complement EIS with APLTK, which establishes heterogeneity and allows for tool support.

In the following, we examine APLTK which has been designed to be tool-kit for agent-oriented platforms, which, on top of that, is fully EIS-compatible. The motivation of this toolkit stems from several aspects. First of all, we intend to establish heterogeneity. Secondly, we intend to move agent-platforms and also agent-programs implemented in different agent-programming languages on a level that allows for different means of comparison. Finally, we aim at proposing interface-standards both for agent-interpreters and auxiliary-tools, which of course corresponds to the overall goals of this very thesis.

Our first intention is to provide the necessary groundwork for the implementation of APLTK. Similar to our approach to EIS we define a set of principles, which in turn are used to specify the design of an infrastructure, which again is assumed to lead to a solid implementation.

9.1. Principles

We define and take into account five principles that are supposed to be the basis for the infrastructure of APLTK:

- **Plug-in architecture:** The overall-infrastructure consists of a *core*, and of *components*, that can be plugged in. Components are 1. *interpreters*, that load, manage and execute agents, 2. *environments* to which agents are connected, that provide them with percepts and in which agents can act, and 3. *tools* that evaluate the execution of multi-agent systems.

- **Minimal assumptions about components:** We assume almost nothing about agents, except that they conform to the agent-definition by Russel-Norvig [55], and that the mental-attitudes can be accessed from the outside and are mappable to a common representation. We assume almost nothing about the interpreters, except for the requirement that each interpreter executes the multi-agent system in a step-wise manner and that each interpreter adheres to a specific interface definition. We assume almost nothing about the tools, except for the requirement that each tool needs to adhere to a specific interface definition. And finally, we

assume almost nothing about the environments, except for the requirement that each environment adheres to the specific interface definition EIS.

- **Execution fairness:** We assume that the executions of the different components are strictly separated and interleaved. That is, interpreters are executed first and in a single-threaded manner. After that the tools are executed. Tools and interpreters should not interfere with the execution of other components. This is undesired because it leads to disturbances when it comes to performance measurements.
- **XML as configuration language:** We use XML as a language for configuring because of its usability, readability and wide use.
- **Areas of application:** We intend to use the toolkit for *profiling*, *automated testing*, *debugging*, *gathering statistics*, and *establishing heterogeneity*.

Based on these (design) principles we now create an infrastructure for heterogeneous multi-agent systems with tool support.

9.2. Infrastructure

In this section, we define APLTK's infrastructure, which is supposed to be faithful to the previous principles. We examine its components and elaborate on their interactions. The appendix of this thesis contains a view on the respective Java interfaces. We consider the APLTK infrastructure to consist of these structures:

- **Interpreter:** We consider an interpreter to be a software component that instantiates and contains agents, which are scheduled for execution and executed by some means provided by that component. It is assumed that each individual agent is a piece of software that processes percepts and generates actions. And finally, each interpreter provides means to query the mental state of each agent.
- **Environment:** Each environment is supposed to be faithful to the notions reflected by EIS. It contains controllable entities that generate percepts and process actions. On top of that, means for querying the environment and the controllable entities are provided.
- **Tool:** We assume that tools are capable of querying interpreters (and their hosted agents) and environments when they are given processing time. Additionally, tools have access to step results, that is the outcome of each step that an interpreter performs as a whole.
- **Core:** This component is a static glue component that facilitates loading, executing and releasing components, which can be dynamically linked during runtime. The core also delivers step-results (amongst other things: execution time, changed mental attitudes) to the tools. The core, can be controlled, that is the overall execution can be paused, resumed or finished.

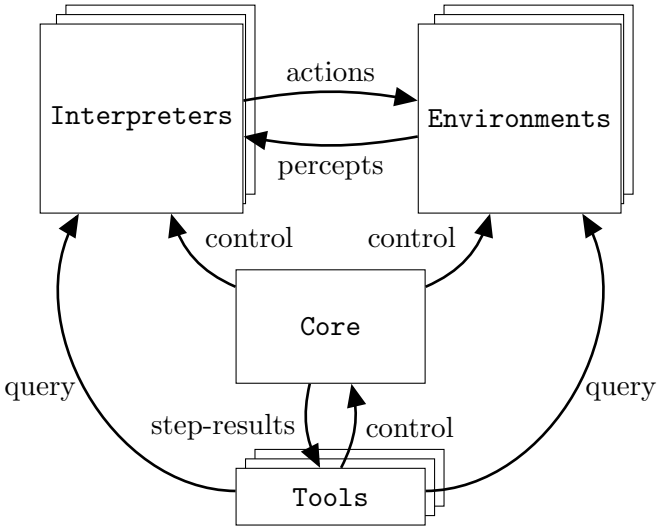


Figure 9.1.: The APLTK infrastructure. Tools are active components, when it comes to querying, but it is the interpreters and environments that provide the data.

Figure 9.1 shows the infrastructure of APLTK. The set of components is constituted by interpreters, tools, and environments. We now have a look at the interactions between individual components. It has been assessed that there is no direct interaction between one interpreter and a second one, between one environment and another and between one tool and a second one. Interpreters and environments interact by relaying actions and percepts as defined by the EIS contract. There is a fixed language for that kind of interaction, which is called the IILang. The core component can control the execution of all interpreters and environments. By control we mean manipulating the state of execution, that is starting/stopping and pausing/resuming. On top of that the core is capable of retrieving step results from all interpreters and relay them to the tools. Each tool, however, is capable of controlling other components indirectly via the core. This allows the user to control their execution in a straightforward and elegant manner. And finally, each tool can query all interpreters and environments.

Up to now, we have elaborated on the components. Now we are about to define an execution model that executes the overall multi-agent system, while being faithful to the principles. The execution model is as follows:

1. execute interpreter(s) one step,

2. deliver step-results to the tools and let them evaluate,

3. repeat.

Executing a single interpreter means that it allows each of its agents to perform one step of their deliberation cycles. This requirement has been introduced in order to ensure both fairness and precision. The subsequent execution of individual agents would allow a

precise measurement of deliberation time, which alternatives like multi-threading would hinder to a significant extent. The execution of one step yields step results. That is the outcome of a single step, consisting of queries and updates of the mental state. Such results are then relayed to the tools which are be capable of taking them into account, while also being able to access the agent's internals at any time.

At this point it is time to accentuate the usefulness of step results. Imagine that a comparative framework should be established that allows to assess the similarity/equivalence of different agents, possibly executed via different interpreters. Now we want that comparison to be based on the number or even the types of mental state updates and queries. Such a framework would require the availability of a record the mental state dynamics actions. If such a record would be available together with an precise time measurement, a good means to compare would be available.

9.3. Interpreter

This section's content is the focus on the interpreter component. Again, an interpreter is a component of our multi-agent system set-up that hosts and executes agents. Firstly, we have a look at the data structures which facilitate the communication of interpreters with other components. Secondly, we elaborate on a general interface for interpreters.

9.3.1. Data Structures

We consider tokens of data that are yielded and relayed by the interpreter and by the agents, during the execution of the overall system. Beliefs and goals are essential:

- `apltk.interpreter.data.Belief` represents an abstract belief, that is something that an agent believes to be true about the world.
- `apltk.interpreter.data.BeliefQuery` denotes a query of the belief base, that is checking if some belief in true with respect to the belief base that an agent currently has.
- `apltk.interpreter.data.BeliefUpdate` is an update of the belief base. This is the addition or the removal of some given belief.
- `apltk.interpreter.data.Goal` represents an abstract goal, that is some state of the world the agent wants to bring about.
- `apltk.interpreter.data.GoalQuery` is a query of the goal base, which means checking of some goal is true with respect to the goal base.
- `apltk.interpreter.data.GoalUpdate` is an update of the goal, that is either the removal or the addition of a given belief.

Beliefs and goals are abstract objects that are defined without any assumptions about an underlying knowledge representation language. Thus in APLTK it is possible to use

any of such languages as long as its expressions can be represented by plain strings. In order to make things easier for us while keeping in mind that we are mostly interested in logic-based approaches, we also provide logic-based beliefs and goals:

- `apltk.interpreter.data.LogicBelief` is logic-based belief, that is an expression in a logic-based knowledge representation language that represents some token that the agent beliefs to be true about the world.
- `apltk.interpreter.data.LogicGoal` is logic-based belief, this means a logic-based expression that represents something that the agent intends to achieve in the world.

To complete the overall picture we also define a couple of objects that go beyond beliefs and goals:

- `apltk.interpreter.data.Plan` represents a plan, which is a sequence or recipe of statements that if executed leads to the achievement of a goal.
- `apltk.interpreter.data.Percept` is a percept, an object that is sent from the environment in order to inform about its current state and/or a change of its state.
- `apltk.interpreter.data.Action` is an action, which is an object that is sent to the environment in order to change the state of the external environment.
- `apltk.interpreter.data.Event` represents an internal event, an object that is issued when some internal situation arises.
- `apltk.interpreter.data.Message` is a message either sent to or received from a second agent.
- `apltk.interpreter.data.BreakPoint` represents that a specific break point has been reached during the agents execution.

Now, with these data objects assumed to be in place we can continue and define a general interface for interpreters.

9.3.2. Interpreter Interface

In order to define a general interface for interpreters that encapsulate and execute agents, we only impose minimal assumptions about agents. We only assume that each agent has a name and on top of that features a set of queryable properties constituted by a subset of belief base, goal base, plan base, event base, percept base and message box. Note, that we do not demand that all agents have all queryable properties. We rather expect that, if the user intends to compare two agents, it has to be decided which subset of those properties is supposed to be the basis of a comparison.

Now, we have a look at the interpreter interface, which imposes a minimal contract for defining and distributing an interpreter:

- `void init(Element parameters)` initializes the interpreter. The parameter is a XML-object. Its structure is specific to the interpreter and is expected to vary between different interpreters.
- `void addEnvironment(EnvironmentInterfaceStandard env)` attaches an environment to the interpreter, which is supposed to be an EIS-compatible environment interface.
- `StepResult step()` executes the interpreter one step. It is assumed that after the termination of that very method each agent managed and executed by the interpreter has been given enough time to deliberate.
- `void release()` releases the interpreter and frees its resources.
- `Collection<String> getAgents()` yields the names of the agents hosted by the interpreter.
- `Collection<Belief> getBeliefBase(String agent)` yields the entire belief base of the agent specified by its name.
- `Collection<Goal> getGoalBase(String agent)` yields the entire goal base of the agent.
- `Collection<Plan> getPlanBase(String agent)` yields the whole plan base of the agent.
- `Collection<Event> getEventBase(String agent)` yields the whole event base of the agent.
- `Collection<Percept> getPerceptBase(String agent)` yields the entire percept base of the agent.
- `Collection<Message> getMessageBox(String agent)` yields the whole message base of the agent.
- `void setBasePath(String basePath)` is a convenience method, that sets up the base bath for loading interpreter-specific files, including agent files and multi-agent systems definitions.

It is assumed that the collections representing a specific base, like the event base, is empty if the corresponding notion is not supported. After creating an interpreter object the `init`-method is called which sets the interpreter up. That is, loading and initializing agents, setting up the inter-agent connections and dependencies and setting up the agent scheduling mechanism. After that environments can be added which are then internally connected to the loaded agents. The querying methods on the other hand, are invoked during the execution of the overall system by connected tools. In the following, we have a look at these tools.

9.4. Tool Interface

Now, we give an overview of a *tool interface* that is supposed to act as a template for specialized tools that can be used for heterogeneous multi-agent systems, and that preferably can be used with several different interpreters. We assume that a tool is any software component that is capable of querying or accessing the interpreters, agents and environments and makes use of these capabilities in order to fulfill some specific function or use. On top of that, a tool can evaluate step-results that contain what has happened inside a considered interpreter while it has been executing one step. A tool is initialized via XML and can react to external signals and events, such as the user stopping the execution of the system, and is supposed to provide means for graphical user interaction if required. These are the general tool methods that constitute a contract for interacting with tools:

- `void init(Element parameters)` initializes the tool. The parameters are denoted by an XML-object. Its structure is specific to the tool and is expected to vary between different tools.
- `public void addInterpreters(Collection<Interpreter> interpreters)` connects the tool with the allocated interpreters. Establishes the tool's access to the interpreters.
- `public void addEnvironments(Collection<EnvironmentInterfaceStandard> environments)` connects the tool with an environment, represented by its EIS-compatible environment interface. Allows the tool to access the environment.
- `public void processStepResults(Collection<StepResult> stepResults)` gives the tool the opportunity to evaluate step results, that is the outcome of atomic steps of the interpreter(s).
- `public void release()` releases a tool after it has ceased its usefulness, usually at the end of the overall execution.
- `public void setBasePath(String basePath)` defines a base-path for files that are required for the tool to fulfill its function.
- `public void setCore(Core core)` connects the tool to the core components.
- `public void handleStarted()` is called when the overall execution is started and allows the tool to do something every time the execution is started.
- `public void handlePaused()` is invoked when the system is paused. This allows a tool to react to that event.
- `public void handleFinished()` is called when the overall execution is finished and gives the tool an opportunity to react accordingly.

- `public void handleSignal(String sender, String signal)` is called when some other signal is to be relayed to the tool.
- `public ToolWindow getWindow()` yields the tool's window, which is a graphical user interface.
- `public String getName()` yields the tools name.

In summary, tools have access to interpreters and environments, while also having the capability to react to state changes of the overall system's execution. The last component that is now left missing from our picture is the core, which connects all other components in a manner that makes sense.

9.5. Core Implementation

As already mentioned, the core is the glue component of APLTK that facilitates the interaction of all other components. The core is responsible for instantiating everything. If it is required to perform several executions of a set up, which might be the case if the user wants to run multiple simulations at a time, the core provides methods to do so. Also, it is the core which is capable of measuring the time that interpreters spend on deliberation.

The moment the core component loads several interpreters and at least a single environment, heterogeneity is established, which is one of the goals of this thesis. After loading, the core executes a main loop that handles all components. A simplified version of that very loop is shown by Algorithm 5.

Right in the beginning, all tools are instantiated and initialized. This is required as one of the first steps, because it is desired that tools can monitor several subsequent runs of the multi-agent system. It can be specified on initialization how often a heterogeneous multi-agent system can be executed. In the regular case a system is run exactly once, but multiple executions are possible. The number of executions is denoted by the variable *runs*. For each run all specified interpreters and environments are instantiated and initialized. After that all necessary connections between the components are established. The run has a state, which is represented by the variable *state*. Its initial value is *running*, which leads to an immediate execution of the multi-agent system. The run ends once the $finished$ state is reached. If the state, however, is not *paused* the following happens: Each interpreter is executed one step while the step results are gathered. After stepping the interpreters all tools are granted access to the gathered step results. If the state is *paused*, which can be the result of some internal or external (user) event, nothing happens. Once the execution is over all interpreters and environments are released. Finally, after all runs, the tools are released as well.

9.6. Summary

In this chapter, we gave a brief overview of APLTK. APLTK is a toolkit for agent programming languages, which is an attempt to complement the EIS standardization effort,

Algorithm 5 The Core-loop.

```
instantiate and initialize tools;
for i = 0; i < runs; i := i + 1 do
  instantiate interpreters;
  register all interpreters with all tools;
  instantiate environment(s);
  register all environments with all tools;
  connect all interpreters with all tools;
  set state to running;
  while state is not finished do
    while state is paused do
      do nothing;
    end while
    for all interpreters do
      execute each interpreter one step and gather step-results;
      for all tools do
        evaluate the step-results;
      end for
    end for
  end while
  release interpreters;
  release environment(s);
end for
release tools;
```

which provided a contract for agent-environment interaction, with an effort to standardizing interpreters and tools. An interpreter from our point of view is a software component that encapsulates, schedules and executes a set of agents. A tool on the other hand is a piece of software that is capable of querying interpreters, their agents and environments in order to fulfill a designated purpose. We have shown a running infrastructure that allows the execution of heterogeneous multi-agent systems with tools support, which has been a goal of this thesis.

10. Adapting 2APL, Goal and Jason

The overall goal of this chapter is to connect 2APL, Goal and *Jason* to EIS and APLTK. A connection to EIS allows the platforms to use environments that comply to the outlined standard. A connection to APLTK, however, facilitates that multi-agent systems which are compliant to the platforms can be executed in a controlled and standalone fashion, while allowing access to the multi-agent systems' internals for standardized tools. We consider all six connections, that is { 2APL, Goal, *Jason* } × { EIS, APLTK } and explain the respective details on a level that is on an appropriate distance to the Java code level.

In this chapter, we consider extensions of 2APL, Goal and *Jason* that facilitate EIS-compatibility and make use of EIS's features. We consider, for all three platforms, these issues:

- loading and instantiating environments,
- instantiating agents and associating them with entities,
- associating agents when new entities show up during runtime,
- acting and perceiving, and
- accessing the environment management system.

For 2APL, we examine the new programming constructs that we introduce for managing the set of agents and the agents-entities relation, while respecting the dynamics behind both. Additionally, we briefly consider the switch that allows developers to use both traditional and EIS-based environments. For good measure, we also elaborate on the new MAS specification language that has been developed while working on the EIS-integration.

For Goal, on the other hand, we consider a new MAS language that allows for connecting agents with entities, while instantiating agents if necessary, both at the beginning and during the execution. We also have a look at how Goal makes use of the relevant EIS features.

On top of the task of establishing EIS-compatibility, we also take into account APLTK-compatibility. We consider for each of the three selected agent-programming platforms how it is appropriately adapted, that is how a standalone interpreter, that is the core of the languages and platforms, can be built from the available projects. We consider

- which functionalities each of the three platforms provides that can be exploited,
- how we can facilitate a unified execution mechanism that works for all platforms, and

- how interfacing with the agents and the agent's internals can and is made possible.

Establishing a unified execution mechanism and interfacing with the agents is crucial for the fulfillment of the APLTK principles and reaching the goal of executing agents, which are implemented using the three platforms in a standardized environment. For that purpose, we extend and adapt the platforms where needed. In general, the platforms are required to be extended with some logging mechanism that keeps track of the mental state evolution. Additionally, it is a crucial issue to ensure that the execution policy is the same. This means that in each step of all of the three standalone interpreters we are going to come up with, each agent performs exactly one iteration of its deliberation cycle, one agent after another.

10.1. Adapting and Extending 2APL

Below is an overview on how the 2APL platform is connected to both EIS and APLTK. The main contributions of this section are a new multi-agent system programming language for 2APL, new special actions for using some of EIS's features and an elaboration on how to create a standalone interpreter for 2APL.

10.1.1. EIS Compatibility

Establishing EIS-compatibility for 2APL means several things: Implementing a modified environment loading mechanism that allows for both traditional and EIS-compatible environments, respecting this on the multi-agent system programming level, facilitating communication in both directions, and managing the agents-entities-relationship on the agent programming level.

After adapting and extending, 2APL now makes use of the XML markup language for specifying multi-agent systems. The decision to use XML came from its straightforwardness, usability and the commonness of use. Besides contributing to the EIS compatibility other features are added to the multi-agent system programming language of 2APL.

The first component, that is relevant for the EIS integration, is the `APAPLBuilder` class, which is responsible for instantiating multi-agent systems. We add a mechanism that allows for switching between traditional environments, that is environments that were implemented based on the 2APL-provided environment class, and EIS compatible ones. This is specified via the `eis`-attribute in the `environment`-tag. The modified `APAPLBuilder` works as follows. Firstly and as usual, the respective jar-file is loaded and the environment interface contained therein is allocated. The already mentioned `environment` might contain a succession of `parameter`-tags, which are evaluated and used in the next step. These tags are transformed into key-value pairs that are accepted by EIS's initialization method. If successful, the resulting environment object is stored as a member of the `APLMAS` class, which we consider next.

The class `APLMAS` stores, amongst other things, one or several environments and one or several agents. We add support for environment listening. That is, for each agent an individual agent-listener is allocated that maps incoming percepts-as-notifications to

```
<apaplmas>
  <environment name="massim" file="eis-0.2-acconnector2007.jar"
  eis="yes"/>
  <agent name="bot1" file="bot.2apl">
    <beliefs file="server.pl"/>
    <beliefs file="botcredentials1.pl"/>
  </agent>
  <agent name="bot2" file="bot.2apl">
    <beliefs file="server.pl"/>
    <beliefs file="botcredentials2.pl"/>
  </agent>
  <agent name="bot3" file="bot.2apl">
    <beliefs file="server.pl"/>
    <beliefs file="botcredentials3.pl"/>
  </agent>
  ...
</apaplmas>
```

Figure 10.1.: A snippet from an exemplary 2APL MAS file specification.

2APL specific external events, in order to allow each agent to react to such incoming percepts.

The mechanism of external actions, that is actions that change the state of the external environment and/or provide the agent(s) with active percepts, is extended to respect new special actions that allow access to EIS's diverse functions. The special actions are as follows:

- `getFreeEntities` yields a list that contains the identifiers of all entities that are currently not associated to any agent,
- `getAllEntities` yields a list that contains the identifiers of all entities that are currently in the environment,
- `associateWith` associates an agent with an entity, whose name has to be provided as a parameter,
- `disassociateFrom` releases the association between the agent that executes the action and an associated entity, and
- `getAllPercepts` provides an agent with all the percepts that are currently available to all its associated entities.

Of course, if an action is executed by an agent that does not qualify as an external action and that is recognized by the interpreter as such, the standard mechanism is employed that relays the action to the external environment directly.

⟨*addagentsaction*⟩ ::= "`addagents`""(" ⟨*ident*⟩ "," ⟨*ident*⟩ "," "`.apl`"", " ⟨*num*⟩ "," ⟨*envs*⟩ "," ⟨*var*⟩ ")"

⟨*envs*⟩ ::= "[]" | "[" ⟨*ident*⟩ ("," ⟨*ident*⟩)* "]"

⟨*terminateaction*⟩ ::= "`terminate`"

Figure 10.2.: 2APL grammar extensions for the multi-agent system dynamics actions.

The pre-EIS implementation of the 2APL platform did not support multi-agent system dynamics, that is the set of agents is specified on the multi-agent programming level and does not change during runtime. This, however, implies that when a new entity shows up while the multi-agent system is executing it is only possible to associate that very entity with an already existing agent. Of course, it is considered to be more straightforward to instantiate a new agent and associate it with the new entity. We satisfy this desire and remove this pre-EIS's restriction by introducing two new basic actions, which we call *multi-agent system dynamics actions*:

- the *add-agents action* allocates new agents and adds them to the multi-agent system, and
- the *terminate action* allows an agent to remove itself from the multi-agent system.

We now elaborate on the syntax (see Figure 10.2) and semantics of the multi-agent system dynamics actions. The syntax of the add-agents action is as follows:

```
addagents(name, filename, qt, envs, result)
```

where

- `name` is the base-name that is used to generate unique names for the agents,
- `filename` is the file from which the agent(s) should be loaded,
- `qt` defines the quantity of instantiated agents,
- `envs` is a list (Prolog) representing the environments in which the agents should be situated, and
- `result` is the result value of the action. It is either the constant `fail` or a list (Prolog) of identifiers containing the unique names of the agents that were created.

The `addagents`-action loads an agent specification from a file, instantiates one or several agents, associates it/them with the given environment(s) and returns the unique

names of the new agent(s). Furthermore each new agent is informed about the name of its creator by an external event `createdby(<name>)`.

We consider this example for adding agents during runtime:

addagents(sally, "sally.2apl", 3, [blockworld], R)

After successfully executing this action, three new agents are situated in the *blockworld*-environment and the result value is $R = [sally1, sally2, sally3]$, which then could be used for communication. Since communication is based on the names of agents the result-value is useful. The action makes an agent aware of the names of the agents that it has created. The same fact holds vice versa: agents that have been created are provided with the information who their creator is. A possible use for this could be, for example, to pass initial beliefs and goals from the creator to the created agents, based on the creator's assessment of the current situation and the overall desires of the set of agents.

The `addagents` action provides each agent with the capability to create new agents, which could then associate themselves with entities, thus solving the problem of handling new entities appearing during runtime. This solution, on the other hand, is not restricted to a fixed mechanism and instead can be handled on the agent-programming level, which opens a wide range of possibilities to solve this issue. For example, each new agent can pro-actively find out which entities are currently available and associate itself with it based on its type. Alternatively the creator-agent could provide the created agent with the information and ask it to form the association, which the creator has decided.

Next, and to obtain functional completeness, we consider the parameter-less action `terminate`. The `terminate`-action immediately halts the agent's execution, removes it from the multi-agent system and releases all its resources. We have considered that it would be fair and useful to inform both the creator and the created agents about the others, but the decision whether to inform the creator about the termination of one of the created agents should be situated in the field of responsibility of the agent that decided to terminate itself. No event is being raised, and if the creator needs to be informed, the agent can send a message. We have decided that an agent can only terminate itself, rather then being terminated by others. This decision has been made in order to maintain the autonomy of each agent involved in the multi-agent system. If the latter feature would be possible, consistency issues would be raised: it is not desirable to let an agent terminate another one while a third one is waiting for a task to be solved. Note that it is always possible for another agent to request an agent to terminate itself by sending a message, but – keeping the property of autonomy in mind – the agent might decide not to terminate itself.

10.1.2. APLTK Compatibility

APLTK-compatibility is rendered possible by a class called `APAPLInterpreter`. This class refers to an instance of `APLMAS`, which is an object-representation of an entire multi-agent system consisting of a set of agents/modules, a set of environment interfaces, a messenger that is responsible for inter-agent communication and an executor, which schedules the execution of the individual agents. The adapted interpreter is capable

of retrieving the names of the instantiated agents, maps 2APL beliefs and goals to the shared data structures. Additionally, it takes into account initialization parameters. The user can specify the multi-agent system file to be loaded. Also, the executor, that is the component that schedules and executes the individual agents, can be defined. And finally, the means for inter-agent communication, that is Jade or not, can be set up. After parsing and interpreting the parameters the multi-agent system is initialized using the multi-agent system file, the executor and the messenger. In 2APL agent scheduling and inter-agent communication are parameters that are not on the multi-agent system level.

We have added the implementation of an executor `SingleThreadedExecutor`, to complement the already existing `MultiThreadedExecutor`. While the latter one executes each agent in a single thread for the sake of fairness, the first executes them sequentially for the sake of reduced overhead and increased control over the execution itself. Each step of the single-threaded involves each agent being allowed to execute its deliberation cycle exactly once. The order of execution is derived from the order the agents are specified in the multi-agent system file.

For agent inspection we have implemented a `Logger` class. This class monitors, during the execution of the multi-agent system and for each individual agent, all belief updates, belief queries, goal additions, goal removals and goal queries for instant inspection and later evaluation.

10.2. Adapting and Extending Goal

Now we inspect how the Goal platform is adapted and extended to provide support for EIS and APLTK. The main contributions are a new multi-agent system programming language that makes use of the most essential of EIS's features, and an elaboration on how a standalone interpreter for Goal multi-agent systems has been implemented.

10.2.1. EIS Compatibility

In order to establish EIS compatibility for the Goal platform, three things have to be coped with. Firstly the connection between Goal and EIS has to be established, replacing Goal's old environment connection. Secondly a new programming language for the multi-agent systems level has to be implemented, that exploits the most of EIS's features. Third and finally, changes in deliberation are necessary for creating a smooth integration.

Extending or replacing the multi-agent programming language becomes necessary because of the requirement to be able to handle environment dynamics, that is appropriately reacting to new entities in the environment. Goal's multi-agent programming language without adaptations/extensions is limited in that respect and only allows static systems, which means that the set of agents is defined at compile time and not at runtime as it is now required. On top of that, EIS installs with the agents and entities idea a new mechanism that needs to be handled appropriately. In the following we consider Goal's new multi-agent programming language.

Figure 10.3 shows the grammar of GOAL's new multi-agent programming language. Requirements fulfilled by that very language are as follows:

- adding an EIS-compatible environment to the multi-agent system,
- separating agent names and agent filenames in order to be able to instantiate several agents from one and the same agent program,
- capability to select for each agent the knowledge representation language,
- associating agents with entities, both at the beginning of execution and during runtime when new entities appear or existing ones are freed from the agents-entities-relation, and
- intuition and readability of the language.

Now, we elaborate on syntax and semantics of the new multi-agent programming language. An arbitrary multi-agent system program specification in GOAL consists of three separate sections:

1. an *environment-description* section that defines the connection to one environment-interface, which is assumed to be compatible to EIS,
2. a section of *agent-files*, that defines a list of GOAL agent program files, and
3. a *launch-policy* section that defines a policy how and when to instantiate agents from these GOAL agent program files.

An environment-description is the starting point for instantiating an environment interface and adding it to the multi agent system. Following the assumption that the environment interface is contained in a jar-file it is scheduled for instantiation like this:

```
environment:  { "environment.jar" . }
```

Of course, a mechanism for using environment initialization parameters is also available. The `environment` statement can be annotated with an `init` tag, which contains the parameters as key-value pairs. Here is an example:

```
environment:  { "environment.jar" .  init [keyA=1,keyB="val"] .  }
```

On the other hand, the agent-files section specifies the set of GOAL agent program files that are to be used in the multi-agent system. It is intended to reference to these files when instantiating agents. GOAL agent program files can be included like this:

```
agentfiles:  { "agent.goal" .  }
```

Individual GOAL agent program files can be annotated. It is possible to specify the knowledge representation language the agents can use, and it is possible to provide a shorthand token that represents the file. If such a shorthand is not provided, the filename without its file-extension is used to fulfill this purpose. An exemplary agent-files section that makes use of these annotations could look like this:

masprogram	::=	[*envdesc*] *agentfiles launchpolicy*
envdesc	::=	**environment: {** *path* **.** [*initParams* **.**] **}**
path	::=	any valid path to a file in quotation-marks
initParams	::=	**init** [*key* = *value* {**,** *key* = *value* }*]
key	::=	*id*
value	::=	*number* | *id*
agentfiles	::=	**agentfiles: {** *agentfile* {**,** *agentfile*}* **}**
agentfile	::=	*path* [*agentparams*] **.**
agentparams	::=	[*nameparam*] | [*langparam*] | [*nameparam* **,** *langparam*] | [*langparam* **,** *nameparam*]
nameparam	::=	**name** = *id*
langparam	::=	**language** = *id*
launchpolicy	::=	**launchpolicy: {** { *launch* | *launchrule* }* **}**
launch	::=	**launch** *agentbasename* [*agentnumber*] **:** *agentref* **.**
agentbasename	::=	***** | *id*
agentnumber	::=	[*number*]
launchrule	::=	**when** *entitydesc* **do** *launch*
entitydesc	::=	[*nameparam*] | [*typeparam*] | [*maxparam*] | [*nameparam* **,** *typeparam*] | [*typeparam* **,** *nameparam*] | [*maxparam* **,** *typeparam*] | [*typeparam* **,** *maxparam*]
typeparam	::=	**type** = *id*
maxparam	::=	**max** = *number*
id	::=	and identifier starting with a lower-case letter
number	::=	a natural number

Figure 10.3.: Grammar of Goal's replacement for the old multi-agent system programming language.

```
agentfiles {
  "agent1.goal" [language=swiprolog,name=file1] .
  "agent2.goal" [language=pddl,name=file2] .
}
```

This specifies two agent files. The first is referenced by the label `file1` and uses SWIProlog as the knowledge representation language. The second is referenced by `file1` and uses PDDL.

The launch-policy section is the final section of the multi-agent program and consists of a list of *launches* and *launch-rules*. A launch is a statement that is executed before actually running the multi-agent system and instantiates agents that do not have a connection to the environment. Here is an example:

```
launchpolicy:  { launch agent1:file1 .  }
```

This instantiates a single agent and uses the agent-file that is referenced to by the token `file1`. It also uses the identifier `agent` as the base-name for the generation of unique agent names. Several agents, however, can be instantiated with a number annotation like this, which would instantiate three agents:

```
launchpolicy { launch agent[3]:file1 .  }
```

The currently used unique-names generator would yield these agent-names: `agent`, `agent1`, and `agent2`.

A launch-rule, on the other hand, is an extension to rules and is applied to instantiate an agent or agents when an entity is available. Additionally, it can also be specified that such an entity qualifies to be associated with an agent, when it matches certain criteria. A launch-rule can be considered to be a launch with a precondition. We consider this straightforward launch-rule:

```
launchpolicy { when entity@env do launch agent:file1 .  }
```

Its interpretation is: whenever there is an available entity, create an agent with the base-name `agent`, while using the agent-program file `file1`, and associate it with the entity.

In order to synchronize an agent's and an entity's name, the asterisk can be employed. The asterisk reflects that an entity's name should be used as the base name for the agent's one, like in this example:

```
launchpolicy { when entity@env do launch *:file1 .  }
```

Also, to remain faithful to EIS's agents-entities principles, several agents can be instantiated while associating them with one and the same entity, like in the following example where three agents would be instantiated and associated with a single entity:

```
launchpolicy {
  when entity@env do launch agent[3]:file1 .
}
```

Now, we consider refined preconditions for launch-rules. First of all, it can be specified to instantiate specialized agents for specific entities, depending on the type of the latter. The following launch-rule would only fire if an entity with type `type1` is available:

```
launchpolicy {
  when [type=type1]@env do launch elevator:file1 .
}
```

When facing a big amount of entities, the amount of agents, which are instantiated can be restricted, by setting an upper limit. The following launch rule would only be fired at most 20 times:

```
launchpolicy {
  when [type=type1,max=20]@env do launch elevator:file1 .
}
```

Exact entity names, or course, can also be taken into account. The next launch rule would only be applied if an entity with the name `entity` is available:

```
launchpolicy {
  when [name=entity1]@env do launch elevator:file1 .
}
```

To conclude this section, we consider two examples. A minimal multi-agent system specification, which would just instantiate a single agent without an environment and thus without any entity association, would look like this:

```
agentfiles: {
  "agentfile.goal"
}
launchpolicy: {
  launch agent:agentfile .
}
```

Finally, below, we show a more sophisticated example. It specifies the instantiation of agents from six different agent-files. This is done while associating each agent with one of the available connectors to the Multi-Agent Programming Contest 2007 scenario:

```
environment: { "eis-0.2-acconnector2007.jar" . }

agentfiles {
  "agent1.goal" .
  ...
  "agent6.goal" .
}

launchpolicy {
```

```
    when [name=connector1]@env do launch agent1:agent1 .
    ...
    when [name=connector6]@env do launch agent6:agent6 .
}
```

After elaborating on the new multi-agent programming language, which respects EIS's agents-entities-principle, we briefly consider the implementation side of the story, which also yields some useful insights.

Two classes are responsible for synchronizing agents with the environment by facilitating acting and perceiving: the `Scheduler` and the `GenericScheduler`. The class `GenericScheduler` implements a round-robing scheduling mechanism that executes the agents. More importantly it also handles the agent dynamics, that is agents being added during runtime.

The class `DefaultEnvironmentInterface` on the other hand encapsulates an instance of our `EnvironmentInterfaceStandard` and calls its method when and where required. Perceiving means to return a default value if there is no valid connection to the environment. If there exists a valid connection, perceiving is handled as expected. Also, an action is not executed if the valid connection is lacking. If this is not the case the action is executed by invoking the respective method. As a reminder, the return value of `performAction` is a map. The values of this very map are then passed to the respective agent, which adds the percepts to its percept buffer. Also, the `DefaultEnvironmentInterface` handles the diverse exceptions that might be raised during execution.

Beyond that, the `RuntimeEnvironmentListener` class notifies attached observers if the environment changes its state. Also, if an entity is freed, its formerly associated agent is deallocated and launch rules are applied in order to instantiate a new agent if possible. Every time a new entity is created and this event is signaled, the launch rules are applied as well. If an entity is reported to be deleted, however, the associated agent is removed from the multi-agent system. Percepts-as-notifications when received are added to percept buffer(s) of the respective agent(s).

Finally, the first and most important task of the `RuntimeServiceManager` is to create the multi-agent system. The class maintains references to all agents and the environment interfaces. Also it handles launching environment interfaces, that is firstly the interface is instantiated, secondly initialization parameters are passed if available, and thirdly already available entities are determined and launch rules are applied to instantiate the initial set of agents.

10.2.2. APLTK Compatibility

The first step to establishing APLTK compatibility is the creation of an adapted scheduling mechanism, provided by the class `SteppingScheduler` which extends GOAL's class `GenericScheduler`. This new scheduler overrides the agent execution mechanism. It selects the set of agents scheduled for execution and executes those agents one by one.

The new `GOALInterpreter` class completes the APLTK compatibility. Firstly the class retrieves an instance of the platform manager, that houses the multi-agent system, and

secondly it instantiates the aforementioned stepping scheduler. For querying purposes the agents' names are retrieved by accessing the runtime scheduler. Querying goals and beliefs is made possibly by getting the agents from the runtime scheduler and evaluating their belief- and goal-bases. Accessing the message boxes and percept bases is equivalent.

The initialization of the Goal interpreter is as follows. In the first step, the multi-agent system file is loaded to set up the multi-agent system. In the second step, the communication middleware is installed. The user can specify to use either Jade, Java RMI, or local. After that, in the third and final step the agents are started.

10.3. Adapting and Extending Jason

The third important section of this chapter explains how *Jason* is connected both to EIS and APLTK. About the first connection we only report on the solution of the problem, since the author of this thesis has only been marginally involved.

10.3.1. EIS Compatibility

Jason's EIS-compatibility is facilitated by three classes:

1. the `Translator` supports mapping IILang expession to *Jason* ones and vice versa,
2. the `JasonAdaptor` facilitates the distribution if EIS-compatible *Jason*-interfaces, and
3. the `EISAdapter` ensures that EIS-compatible interfaces can be used by the *Jason* runtime.

The `JasonAdapter` encapsulates an arbitrary *Jason* environment and makes it EIS-compatible, mostly by relaying method calls. This class is an abstract one, which has to be customized for each *Jason* environment that is supposed to be equipped with EIS-compatibility. For initialization, each provided IILang parameter is mapped to a Prolog expression, after which the environment is set up. Killing the EIS environment interface simply stops the *Jason* environment. Acting is facilitated by relaying the action-to-be-performed to the *Jason* environment. Perceiving means getting the current percepts. If there are no current percepts, the previous percepts. If there are also no previous percepts an empty list is returned.

The `EISAdapter` encapsulates an arbitrary EIS-compatible environment interface and facilitates communication in both directions. Firstly, for initialization, it handles parameters provided in the MAS file. In the very first step, the jar-file that contains the environment interface is loaded. After that, a custom environment listener is attached. This listener only reacts to the state change of the environment interface, but not to the addition of an entity, the deletion of one, or the freeing of another. Then, the parameters provided in the MAS file are mapped to IILang parameters for calling the initialization method of the environment interface. This includes parameters for associating agents and entities and attaching appropriate agent listeners (see Figure 10.4). If, on the other

```
environment: jason.eis.EISAdapter(
  "lib/eiscarriage-0.3.jar",
  agent_entity(robot,robot1),
  agent_entity(robot,robot2)
)
```

Figure 10.4.: Snippet of a *Jason* MAS file. It shows how the connection to an EIS-compatible environment interface is established, while the agent `robot` is associated to the two entities `robot1` and `robot2`.

hand, no agents-entities-association is specified in the MAS file, each agent is associated to the entity with the same name. Only after that the environment interface is initialized with the provided parameters. In the final step, the environment is started by calling the respective method. Percepts that arrive as percepts-as-notifications are added to the agent right away. Getting all percept does the same but on top of that also annotates the percepts with tokens similar to `entity(entityName)`, that denotes the source of the percept(s). For executing actions an entity name can be specified as second parameter next to the action itself.

10.3.2. APLTK Compatibility

Establishing APLTK compatibility for *Jason* boils down to four essential steps:

1. adding a logger that facilitates keeping track of the agents' mental states' evolution,
2. extending the default belief base to make use of that logger,
3. creating a custom agent architecture that ensures access to the agents' internals, and
4. setting up an APLTK-compatible interpreter that employs all the mentioned components.

The `Logger` class keeps track of added beliefs, removed beliefs, queried beliefs, added goals, removed goals, and queried goals. It implements the `GoalListener` interface in order to keep track of failed, finished and started goals. All of those data-tokens are stored in separate collections, that can be accessed at any time when required.

Added on top is the `LoggingBeliefBase` class that extends the default belief base and makes use of the aforementioned logger. The class keeps track of added, removed and queried beliefs.

The constitution of an individual agent, that allows access to its internals and the evolution of those, is facilitated by the `AgentArch` class. This component is provided in the *Jason* software package. This agent architecture creates an agent from a specified agent program, attaches the described, specialized belief base and a goal listener for

monitoring the goals. Additionally, the agent architecture includes a straightforward message-box implementation for inter-agent communication.

The core class of the APLTK-compatible interpreter is the `JasonInterpreter`. This class stores individual agents in the form of a map that maps agent names to agent architectures. It parses a specified multi-agent system file and instantiates the agents, that is the agent architecture class. On top of that, the interpreter provides functionality to extract a goal-base from the set of intentions and the set of events. On the level of abstraction that corresponds to our research such means are possible. We straightforwardly assume that events encode goals-to-be-handled and intentions are goals that are currently being handled. Finally, an execution of a single step of the *Jason* interpreter is equivalent to executing the deliberation cycle of each agent exactly once, followed by evaluating the logs of each agent, which are then made available for later evaluation by appropriate tools.

10.4. Summary

In this chapter we have taken into account 2APL, Goal and *Jason* and have, for each platform, established compatibility to both EIS and APLTK.

11. Agents Computing Fibonacci Numbers

In this chapter, we provide the results of a case-study that shows the applicability and usefulness of APLTK. We build directly on the adaptations of the three agent-programming platforms 2APL, GOAL and *Jason*, that we have investigated in the previous chapter (see Chapter 10). Now, we use APLTK to compare three similar agent programs, written in 2APL, GOAL and *Jason*, that compute Fibonacci numbers in a sequential manner. Our experimental set-up is constituted by the three agent interpreters. For each platform we employ a single agent that computes Fibonacci numbers. On top of that we make use of a tool that measures the time consumed by the computation. In addition we inspect agent-traces and elaborate on their equivalence/similarity.

11.1. Set Up

In this section, we elaborate on a case study that is based on a very simple task: computing Fibonacci-numbers [48]. As a reminder, here is the definition of the Fibonacci sequence:

$$F_1 := 1,\ F_2 := 1,\ F_{n+2} := F_{n+1} + F_n$$

Although this scenario is fairly simple, it turns out later that results derived from it are relatively insightful. We inspect agent programs for 2APL, GOAL and *Jason*, that compute the first thousand Fibonacci-numbers. We execute those agents and then compare how long it takes each program to reach that goal. While executing we take samples.

To facilitate our experiments, we make use of APLTK. We use the APLTK-compatible interpreters for 2APL, GOAL and *Jason*, that we have described in an earlier chapter (see chapter 10). Also, we use a specialized tool, that we call *breakpoint debugger*. That tool hits a breakpoint, that is a sample, on every 50th computed Fibonacci number, and measures and logs the time the agent(s) used for computing so far. After each step of the deliberation cycles, the breakpoint debugger inspects the mental states of the agent(s) and checks if the current breakpoint, that is the next number of interest, is stored as a fact in the current belief base.

We have a look at three agents, one for each agent programming language. The three agents are similar in the aspects, that they all are initialized with the first two Fibonacci numbers and means to compute the remaining others. The Figures 11.1, 11.2, and 11.3 show the source-codes. The agents work as follows:

1. each agent knows the first two numbers right from the beginning,

2. each agent has the initial goal of computing the first one thousand numbers, starting with the third, and

3. if an agent has the goal of computing a specific number, it computes it, stores the result in the belief base, drops the respective goal, and adopts the goal of computing the next one, all until the final number is computed.

11.2. Agent Programs

In an earlier chapter (see chapter 3), we have examined the syntax and semantics of the considered agent programming languages. Now, we only elaborate on those syntactical and semantical notions that are required for understanding the examples as a kind of adequate reminder.

The 2APL agent program in Figure 11.1 shows the basic syntactical components of an 2APL agent. The initial belief base consists of a full Prolog program and thus is usually a list of facts and rules. The belief base is facilitated by JIProlog[1]. In our case it contains only facts about the first two Fibonacci-numbers. The initial goal base consists of a single goal, that is calculating the first one thousand numbers, starting with the third. In general, the belief-updates section contains actions that update the belief base by adding/removing facts if certain preconditions hold. We use a single belief-update to insert computed numbers. PG-rules trigger the instantiation of plans in order to reach goals. We have two rules. The first rule computes a number and triggers the computation of the next one, whereas the second rule computes a Fibonacci-number and ceases computation afterwards. 2APL programs can also contain rules for handling events or repairing plans, but we do not need such rules in this chapter.

The Goal agent program in Figure 11.2 consists of a belief base, a goal base and a program section. Again, the initial belief base is a full Prolog program. Goal uses SWIProlog[2] as a knowledge representation language. Note, however, that other languages can also be used for knowledge representation in Goal. The initial belief base contains two facts representing the first two Fibonacci-numbers and the goal base consists of the single goal to compute the first one thousand Fibonacci-numbers, beginning with the third. In Goal the program-section specifies how the state of the agent changes over time. It contains three rules. The first one drops the goal of calculating a specific number if it is believed by the agent, the second one calculates the n-th number if it is not believed, and the third one raises the goal of computing the next number.

The *Jason* agent program in Figure 11.3 consists of a belief base, a goal base and two rules. The belief base of a *Jason* agent is expressed by a logic-programming-like language, that incorporates facts, rules and strong negation. *Jason* uses a knowledge representation language that has been tailored specifically for *Jason*, instead of encapsulating an already existing one like 2APL and Goal do. Like before, the belief base consists of two facts and the goal base contains a single goal. The first rule calculates the next Fibonacci-

[1] `http://www.ugosweb.com/jiprolog/`
[2] `http://www.swi-prolog.org/`

```
Beliefs: fib(1,1). fib(2,1).
Goals: calcFib(3,1000)
BeliefUpdates: { true } Fib(N,F) { fib(N,F) }
PG-rules:
  calcFib(N,Max) <-
    N < Max and fib(N-1,Z1)
    and fib(N-2,Z2) and is(Z,Z1+Z2)  |
    {[
      Fib(N,Z);
      adopta(calcFib(N+1,Max));
      dropgoal(calcFib(N,Max))
    ]}
  calcFib(N,Max) <-
    N = Max and fib(N-1,Z1)
    and fib(N-2,Z2) and is(Z,Z1+Z2)  |
    {[
      Fib(N,Z);
      dropgoal(calcFib(N,Max))
    ]}
```

Figure 11.1.: The Fibonacci 2APL agent program.

number and triggers the computation of the successive one. The second rule calculates the last number and thus terminates the overall computation.

We have implemented all three agent programs in accordance with two criteria:

1. all agent programs should yield agent runs that are similar (more on that later), and

2. the programs should execute as fast as possible.

It should be noted, that, of course, there are agent programs that perform faster, but would not yield the desired agent run. We could come up with programs that make use of respective language-features that would facilitate a swift computation, but we do not for the reason mentioned before.

11.3. Agent Runs and Similarity

In this part of the chapter, we show and elaborate on the similarity results gained when comparing the three agent programs directly and automatically using APLTK. We inspect the agent runs, not in their entirety, but to an extent that makes our point clear.

This is an excerpt of the generic agent run generated by the 2APL agent:

1. $B = \{fib(2,1)., fib(1,1).\}$
   $G = \{calcFib(3,1000).\}$

```
main: fibonacci {
  beliefs { fib(1,1). fib(2,1). }
  goals { calcFib(3,1000). }
  program[order=linear] {
    if
      goal(calcFib(N,Max)),
      bel(Prev is N-1),
      goal(calcFib(Prev,Max))
    then
      drop(calcFib(Prev,Max)).
    if
      goal(calcFib(N,Max)),
      bel(
        not(fib(N,F)),Prev is N-1,
        PrevPrev is Prev-1, fib(Prev,FPrev),
        fib(PrevPrev,FPrevPrev),
        FN is FPrev + FPrevPrev
      )
    then
      insert(fib(N,FN)).
    if
      goal(calcFib(N,Max)),
      bel(fib(N,F),Next is N+1)
    then
      adopt(calcFib(Next,Max)).
  }
}
```

Figure 11.2.: The Fibonacci GOAL agent program.

2. $B = \{fib(2,1)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(4,1000).\}$

3. $B = \{fib(2,1)., fib(3,2)., fib(4,3)., fib(1,1).\}$
   $G = \{calcFib(5,1000).\}$

4. $B = \{fib(2,1)., fib(3,2)., fib(4,3)., fib(5,5)., fib(1,1).\}$
   $G = \{calcFib(6,1000).\}$

This is an excerpt of the generic agent run generated by the GOAL agent:

1. $B = \{fib(2,1)., fib(1,1).\}$
   $G = \{calcFib(3,1000).\}$

2. $B = \{fib(2,1)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(3,1000).\}$

```
1  fib(1,1). fib(2,1).
2  !calcFib(3,1000).
3
4  +!calcFib(N,Max) :
5    N < Max & fib(N-1, Z1) & fib(N-2,Z2) & Z = Z1+Z2 <-
6    +fib(N,Z);
7    !!calcFib(N+1,Max).
8
9  +!calcFib(N,Max) :
10   N == Max & fib(N-1, Z1) & fib(N-2,Z2) & Z = Z1+Z2 <-
11   +fib(N,Z).
```

Figure 11.3.: The Fibonacci *Jason* agent program.

3. $B = \{fib(2,1)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(4,1000)., calcFib(3,1000).\}$

4. $B = \{fib(2,1)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(4,1000).\}$

5. $B = \{fib(4,3)., fib(2,1)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(4,1000).\}$

6. $B = \{fib(4,3)., fib(2,1)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(4,1000)., calcFib(5,1000).\}$

7. $B = \{fib(4,3)., fib(2,1)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(5,1000).\}$

8. $B = \{fib(4,3)., fib(2,1)., fib(5,5)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(5,1000).\}$

9. $B = \{fib(4,3)., fib(2,1)., fib(5,5)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(6,1000)., calcFib(5,1000).\}$

10. $B = \{fib(4,3)., fib(2,1)., fib(5,5)., fib(3,2)., fib(1,1).\}$
    $G = \{calcFib(6,1000).\}$

This is an excerpt of the generic agent run generated by the *Jason* agent:

1. $B = \{fib(2,1)., fib(1,1).\}$
   $G = \{calcFib(3,1000).\}$

2. $B = \{fib(2,1)., fib(1,1)., fib(3,2).\}$
   $G = \emptyset$
3. $B = \{fib(2,1)., fib(1,1)., fib(3,2).\}$
   $G = \{calcFib(4,1000).\}$
4. $B = \{fib(2,1)., fib(4,3)., fib(1,1)., fib(3,2).\}$
   $G = \emptyset$
5. $B = \{fib(2,1)., fib(4,3)., fib(1,1)., fib(3,2).\}$
   $G = \{calcFib(5,1000).\}$
6. $B = \{fib(2,1)., fib(5,5)., fib(4,3)., fib(1,1)., fib(3,2).\}$
   $G = \emptyset$
7. $B = \{fib(2,1)., fib(5,5)., fib(4,3)., fib(1,1)., fib(3,2).\}$
   $G = \{calcFib(6,1000).\}$

We can see immediately that the belief bases of the agents under consideration evolve in the same way, but the goal bases' evolutions differ greatly. Using our definition from an earlier chapter (see Chapter 5), we conclude that the agents are $n - B, G$-equivalent with $n := 999$, $B := B_A$, and $G := \emptyset$. That is, when filtering the generic agent runs down to ones that only respect the belief base, then the programs are similar for 999 steps, which is the exact number of different steps it takes to compute the first one thousand Fibonacci-numbers.

11.4. Performance Results

Finally, we have a look at how fast the agents compute the first one thousand Fibonacci-numbers. It should be stressed, at this point and for the sake of comparability, that this task is very trivial. But although it is very simple, agents doing that computation perform relatively bad, as compared to other approaches. For comparison, we have computed the numbers using a plain Java-program written from scratch. This program took about $0.361ms$ on our machine[3].

As the Table 11.1 and Figure 11.4 clearly show, the *Jason* agent program performs best, followed by 2APL and Goal. Now, it becomes clear that we need a deeper examination on where the reasons for the differences in performance lie. We suppose that the use of the specific knowledge representation languages play a major role.

11.5. Summary

We have shown a case-study in which we use APLTK for comparing three similar agent programs implemented in 2APL, Goal and *Jason*. We have also shown how one shared

[3]MacBook Pro CoreDuo 2GHz with 2GB RAM running MacOSX Snow Leopard.

	2APL		Goal		*Jason*	
Fib	Step	Time	Step	Time	Step	Time
50	47	752.9	141	968.5	94	27.7
100	97	1226.5	291	2005.6	194	55.1
150	147	1652.7	441	3047.2	294	84.5
200	197	2048.0	591	4080.6	394	111.4
250	247	2457.4	741	5107.0	494	136.4
300	297	2854.0	891	6126.7	594	161.0
350	347	3158.8	1041	7154.2	694	185.3
400	397	3497.4	1191	8171.1	794	208.4
450	447	3885.3	1341	9186.9	894	230.1
500	497	4172.9	1491	10208.5	994	251.2

Table 11.1.: Performance profiles for the three agents, showing the number of steps and the execution time.

tool, that has access to the agents' mental states can be used to facilitate a comparison and measure the performance of the agents.

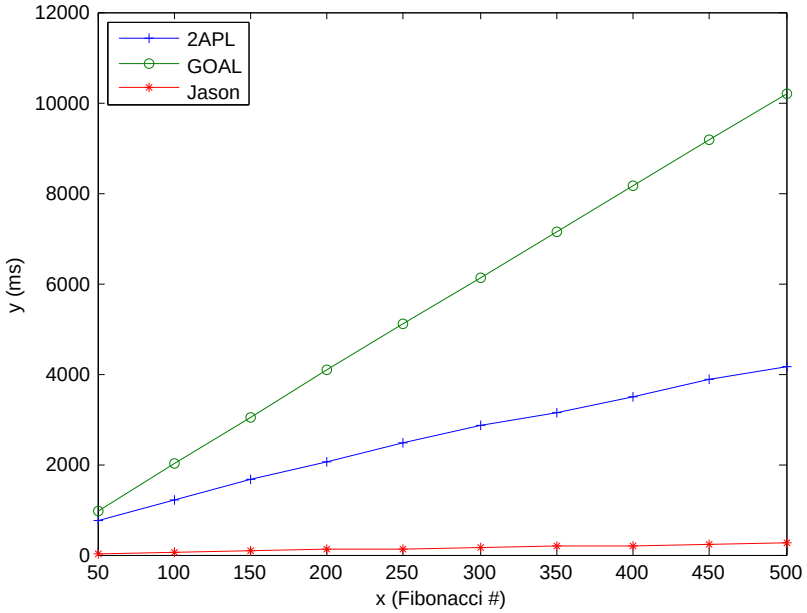


Figure 11.4.: The performance of the three agents. The *Jason* agent is fastest, followed by the 2APL and Goal agents.

12. Summary, Conclusions, Possible Future Work and Acknowledgements

At the beginning of this thesis, we have outlined a set of goals that we intended to reach in the course of this work. Our overall goals consisted of modularity/reusability and establishing heterogeneity. The first one was identifying components of agent-programming platforms that are worth extracting and sharing. This unavoidably would and did lead to standardizing such components on an adequate and least restrictive level of abstraction. Establishing heterogeneity, on the other hand, that is populating a single environment with agents executed by different interpreters, could be established by making use of some of the results of the standardizing effort, which we did. That is, standardizing environments and interpreters, and plugging them into a unified framework that executes them. On top of that, we also established tool support, that allows for reusable tools, that fulfilled another of our requirements.

In this thesis, we have outlined our problem(s) and introduced the necessary background that is required for understanding. We have provided an introduction to BDI programming, that is programming agents which have a mental state, that denotes what the agent knows, wants to achieve and how to achieve his goals. Afterwards, we have provided a formalization of our infrastructure and, based on that, defined a couple of agent equivalence notions. Following this matter, we considered applications of out approaches. We have considered EIS which is an environment interface standard, that facilitates the portability of environments. Then, we have shown an navigation example that uses EIS' separation of agents and entities principle, and that showed how multiple entities can be controlled by a single agent in a navigation scenario. Thereafter, we have elaborated on *EISMASSim*, which is an EIS-compatible environment interface for the Multi-Agent Programming Contest, that not only facilitated the client-server connection which is required for perceiving and acting, but also gathered some statistics. Following this, we have introduced APLTK, which is an agent programming toolkit that complements EIS by defining interface standards for interpreters and tools. Later, we have shown how to establish EIS- and APLTK-compatibility for the BDI based agent programming platforms 2APL, Goal and *Jason*. Finally we have concluded the practical part of the thesis with a case study in which we compared the performance of agents implemented in 2APL, Goal and *Jason*.

This thesis has a couple of weak points. First and foremost it is rather limited, when it comes to the agent software under consideration. We only focussed on three different agent-programming platforms, that is 2APL, Goal and *Jason*, whereas research beyond those three would have been possible. Another weak point is that more experiments would have benefited the overall work a lot. This is especially obvious, when it comes

to the comparison and statistics generation of different agent implementation. On top of that, comparison infrastructure is rather experimental and could be deepened, which also implies extensions of the equivalence notions used in this work. And finally, APLTK or similar framework must be established in the community, as it is now not known.

An overall strong point of this thesis is that EIS has been very well received and used for several platforms and environments. EIS left a good impression and made some considerable impact on the programming multi-agent systems community. A good number of environment interfaces crystallized and it was shown that those could be used easily with established agent-programming platforms. The agent contest, which is also an important aspect of this work, is a well established institution in the community.

In summary, writing this thesis was a very insightful and inspiring endeavor, which fueled a couple of issues that are relevant for the programming multi-agent systems community.

12.1. Possible Future Work

Possible future work includes, but is not limited to, a good continuation of the standardization effort, that has been outlined in this thesis. After working with environments, interpreters and tools, it would be worth to consider also and amongst other things knowledge representation technologies, communication modules and interfaces for accessing legacy code. And, of course, our approaches as a whole could be expanded to more agent platforms. For the agent contest, on the other hand, it would be worthwhile to examine and solve the problem if and how the contest could be used to compare platforms as a whole instead of comparing just individual agent implementations.

12.2. Acknowledgements

Several results of this thesis would not have been possible without the efforts of fellow researchers. Some of the work presented here appeared in other scientific publications. In the following, we acknowledge the cooperation with others and mention where parts of this work have appeared before.

The Multi-Agent Programming Contest, which has been presented in Chapter 2, has been steered and organized by Jürgen Dix, Michael Köster, Peter Novak, Mehdi Dastani, Jomi Hübner and the author. Since its beginning it has generated a steady stream of publications [24, 25, 26, 8, 9, 7].

The comparison of environment development techniques, as presented in Chapter 3, appeared in a couple of publications that contained the EIS standard for agent environment interaction [12, 6, 37, 10, 11]. EIS has been developed by the author, with an strong cooperation with Koen Hindriks and others. Other parts appeared in a technical report [13] about APLTK, which provides standards for tools and interpreters, written by the author and others.

Parts of our formalizations, as shown in Chapter 3, were conceived while working on the environment interface standard together with Koen Hindriks and others.

Parts of the equivalence notions presented in Chapter 5 appeared in an AAMAS paper (not accepted), written by the author together with Mehdi Dastani, Jomi Hübner and Koen Hindriks. The results were later published as a technical report [13], which also contained some of our work on APLTK shown in Chapter 9, and on the Fibonacci experiment shown in Chapter 11. Out of the mentioned authors, Koen Hindriks proved to be a major source for insights and inspiration.

EIS, which has been introduced in Chapter 6, has appeared in multiple publications [12, 6, 37, 10, 11]. It has been developed by the author, who strongly cooperated with Koen Hindriks. Hindriks assumed a very influential position before, during and after development. Wouter Pasman proved to be a very significant source of useful and cherished bug and feature requests, while also significantly contributing to the design of EIS. We also would like to acknowledge the very useful contributions and comments provided by Rafael Bordini, Lars Braubach, Rem Collier, Mehdi Dastani, Jürgen Dix, Jomi Hübner and Alexander Pokahr. And finally the very inspiring comments provided by the students at Delft Technical University were of great help.

The main bulk of the work on potential fields, as shown in Chapter 7, appeared in the papers [5] and [14]. The first publication introduced the idea of one agent controlling several entities. The second publication built on top of the first approach and refined the use of the potential fields method. The paper was co-authored by the author's colleagues Tim Winkler and Randolf Schärfig from the computer graphics group in Clausthal. Mehdi Dastani and Nick Tinnemeier helped a lot with the groundworks for the experiments, that is using 2APL for our purposes. Both provided a lot of inspiration, insights and help, for which the author is very thankful.

The Agents on Mars scenario has been designed and implemented by Jürgen Dix, Jomi Hübner, Michael Köster, Federico Schlesinger and the author. *EISMASSim*, which is shown in Chapter 8, has been designed and implemented by the author with the aid of the aforementioned colleagues. Both the scenario and the environment interface have been used in teaching and in the actual tournament. There have been a couple of publications related to the Multi-Agent Programming Contest in the past [24, 25, 26, 8, 9, 7].

Several people worked on the agent platform adaptations presented in Chapter 10. The author played a major role in establishing EIS compatibility for 2APL with good advice and support kindly provided my Mehdi Dastani, who also took a great part in creating the new multi-agent system programming language and inspired to implement a bridge for traditional and EIS-compatible environments. Establishing APLTK compatibility would have been more difficult without Mehdi Dastani's aid. The author also played a major role in making Goal EIS-compatible. He was mainly responsibly for connecting Goal to EIS on the implementation level. He also took a big part in designing and implementing support for the new multi-agent system programming language that Goal now heavily relies on. These endeavors would not have been possible without the support and help provided by Koen Hindriks and Wouter Pasman. For *Jason* Jomi Hübner established a very straightforward and well-thought-out bridge that connected *Jason* and EIS. His work is reflected in this chapter. The author only played a minor role in this part. Support for APLTK was established by the author with great counsel and good assistance by Mehdi Dastani, Koen Hindriks, Jomi Hübner and Wouter Pasman.

A. APL Syntax Definitions and Environments

In the following, we use the *extended Bachus-Naur form* (EBNF) for specifying the syntax of the different programming languages we have used with in this thesis. The EBNF is a standardized syntactic metalanguage (international standard, ISO 14977) that is useful for defining the syntax of linear sequences of symbols in general, and programming and other languages in particular.

An EBNF-specification of a language consists of a set of production rules. A rule assigns sequences of symbols to a variable symbol. Symbols are variable symbols, e.g. `<Zero>` and `<One>`, and terminal symbols, e.g. `"0"` and `"1"`.

Rule ends with a semicolon, which is the termination symbol. Here are two examples:

```
<Zero> = "0" ;
<One> = "1" ;
```

The first operation is the *concatenation* which joins two items. The word `10` can be generated like this:

```
<Two> = <One>  <Zero> ;
```

Alternatives can be defined using the *separation*-operator. The following rule generates the words `0` and `1`:

```
<Digit> = <Zero> | <One> ;
```

Symbol sequences can be repeated (zero or more). This rule produces the words `0`, `1`, `10`, `11` et cetera, including the empty word:

```
<Number> = { <Digit>  } ;
```

In order to produce symbol sequences repeatedly that are not empty, we use another operation. This rule produces the words `0`, `1`, `10`, `11` et cetera, excluding the empty word:

```
<NumberNotEmpty> = { <Digit>  }+ ;
```

Optional parts of symbol-sequences can be expressed using the options-operation. The following rule would produce the words `1`, `01`, `11`, `101` et cetera.

```
<OddNumber> = [ <Number> ]  <One> ;
```

Parentheses are used to group items together for the sake of clearness:

(<Zero> | <One>).

Finally, it is possible to express special sequences, which are to be interpreted on a meta-level (i.e. interpreted by the reader):

? *a lower-case string* ?

A.1. 2APL Agent Files

A.1.1. MAS-Files

```
<MAS_Prog>      = { <AgentName> ";" <filename> [ <Int> ]
                  [ <Environments> ] }+ ;
<AgentName>     = <Ident> ;
<FileName>      = <Ident> ".2apl" ;
<Environments>  = "@" <Ident> { "," <Ident> } ;
```

A.1.2. Agent files

```
<Agent>         = { "Include:" <Ident>
                | "BeliefUpdates:" <BelUpSpec>
                | "Beliefs:" <Beliefs>
                | "Goals:" <Goals>
                | "Plans:" <Plans>
                | "PG-rules:" <PGRules>
                | "PC-rules:" <PCRules>
                | "PR-rules:" <PRRules> } ;
<BelUpSpec>     = "{" <BelQuery> "}" <BeliefUpdate> "{" <Literals> "}" ;
<Beliefs>       = { <GroundAtom> "." | <Atom> ":-" <Literals> "." }+ ;
<Goals>         = <Goal> { "," <Goal> } ;
<Goal>          = <GroundAtom> { "and" <GroundAtom> } ;
<BAction>       = "skip" | <BeliefUpdate> | <SendAction> | <ExternalAction> |
                | <AbstractAction> | <Test> | <AdoptGoal> | <DropGoal> ;
<Plans>         = <Plan> { "," <Plan> } ;
<Plan>          = <BAction> | <SequencePlan> | <IfPlan> | <WhilePlan> |
                  <AtomicPlan> ;
<BeliefUpdate>  = <Atom> ;
<SendAction>    = "send(" <IdentVar> <IdentVar> <atom> ")" |
                  "send(" <IdentVar> <IdentVar> <IdentVar> <IdentVar>
                  <atom> ")" ;
<ExternalAction>= "@" <Ident> "(" <Atom> "," <Var> ")" ;
<AbstractAction>= <Atom> ;
<Test>          = "B(" <BelQuery> ")"
                | "G(" <GoalQuery> ")"
                | <Test> "&" <Test> ;
```

```
<AdoptGoal>     = "adopta(" <GoalVar> ")"
                | "adoptz(" <GoalVar> ")" ;
<DropGoal>      = "dropgoal(" <GoalVar> ")"
                | "dropsubgoals(" <GoalVar> ")"
                | "dropsupergoals(" <GoalVar> ")" ;
<SequencePlan>  = <Plan> ";" <Plan> ;
<IfPlan>        = "if" <Test> "then" <ScopePlan>
                  [ "else" <ScopePlan> ] ;
<WhilePlan>     = "while" <Test> "do" <ScopePlan> ;
<AtomicPlan>    = "[" <Plan> "]" ;
<ScopePlan>     = "{" <Plan> "}" ;
<PGRules>       = { <PGRule> }+ ;
<PGRule>        = [ <GoalQuery> ] "<-" <BelQuery> "|" <Plan> ;
<PCRules>       = { <PCRule> }+ ;
<PCRule>        = [ <Atom> ] "<-" <BelQuery> "|" <Plan> ;
<PRRules>       = { <PRRule> }+ ;
<PRRule>        = [ <PlanVar> ] "<-" <BelQuery> "|" <Plan> ;
<GoalVar>       = <Atom> { "and" <Atom> } ;
<PlanVar>       = <Plan> | <Var>
                | "if" <Test> "then" <ScopePlanVar>
                  [ "else" <ScopePlanVar> ]
                | "while" <Test> "do" <ScopePlanVar>
                | <PlanVar> ";" <PlanVar> ;
<ScopePlanVar>  = "{" <PlanVar> "}" ;
<Literals>      = <Literal> { "," <Literal> } ;
<Literal>       = <Atom> | "not" <Atom> ;
<BelQuery>      = "true" | <BelQuery> "and" <Belquery>
                | <BelQuery> "or" <Belquery>
                | "(" <BelQuery> ")" | <Literal> ;
<GoalQuery>     = "true" | <GoalQuery> "and" <GoalQuery>
                | <GoalQuery> "or" <GoalQuery>
                | "(" <GoalQuery> ")" | <Literal> ;
<IdentVar>      = <Ident> | <Var> ;
<Ident>         = ? a string starting with an underscore letter ? ;
<Var>           = ? a string starting with an uppercase letter ? ;
<Atom>          = ? a logical atom ? ;
<GroundAtom>    = ? a ground logical atom ? ;
<Belief>        = ? Prolog fact or rule ? ;
```

A.2. Goal

A.2.1. MAS-Files

```
<MASProgram>     = [ EnvDesc] <AgentFiles> <LaunchPolicy> ;
<EnvDesc>        = "environment:" <Path> "." ;
<Path>           = ? any valid path to a file in quotation-marks ?;
<AgentFiles>     = "AgentFiles:" "{" <AgentFile> { "," <agentfile> }
                   "}" ;
<AgentFile>      = <Path> [ <AgentParams> ] ;
<AgentParams>    = "[" <NameParam> "]"
                 | "[" <LangParam> "]"
                 | "[" <NameParam> "," <LangParam> "]"
                 | "[" <LangParam> "," <NameParam> "]" ;
<NameParam>      = "name=" <Id> ;
<LangParam>      = "language=" <Id> ;
<LaunchPolicy>   = "launchpolicy:" "{" { <Launch> | <LaunchRule> } "}" ;
<Launch>         = "launch" <AgentBaseName> [ <AgentNumber> ] ":"
                   <AgentRef> "." ;
<AgentBaseName>  = "*" | <Id> ;
<AgentNumber>    = <Number> ;
<LauchRule>      = "when" <EntityDesc> "do" <Launch> ;
<EntityDesc>     = "[" <NameParam> "]"
                 | "[" <TypeParam> "]"
                 | "[" <MaxParam> "]"
                 | "[" <NameParam> "," <TypeParam> "]"
                 | "[" <TypeParam> "," <NameParam> "]"
                 | "[" <MaxParam> "," <TypeParam> "]"
                 | "[" <TypeParam> "," <MaxParam> "]" ;
<TypeParam>      = "type=" <Id> ;
<MaxParam>       = "max=" <Number> ;
<Id>             = ? an identifier starting wit a lower-case letter ?;
<Number>         = ? a natural number ?;
```

A.2.2. Agent-Files

```
<Program>      = "main" <Id> "{"
                 [ "knowledge" "{" { <Clause> } "}" ]
                 "beliefs" { <Clause> }
                 "goals" { <PosLitConj> }
                 "program" { <ActionRule> | <Module> }+
                 "action-spec" "{" <ActionSpecification> "}"
                 "}" ;
<Module>       = "module" <Id> "{"
```

```
                    "context" "{" <MentalStateCond> "}"
                    [ "knowledge" "{" { <Clause> } "}" ]
                    "goals" { <PosLitConj> }
                    "program" { <ActionRule> | <Module> }+
                    [ "action-spec" "{" <ActionSpecification> "}" ]
                    "}" ;
<Clause>         = ? any legal Prolog clause ?;
<PosLitConj>     = "Atom" { "," <Atom> } ;
<LitConj>        = [ "not" ] "Atom" { "," [ "not" ] <Atom> } ;
<Atom>           = <Predicate> [ <Parameters> ] ;
<Parameters>     = "(" <Id> { "," <Id> } ")" ;
<ActionRule>     = "if" <MentalStateCond> "then" <Action> ;
<MentalStateCond>= <MentalAtom> { "," <MentalAtom> }
                 | "not" "(" <MentalStateCond> ")" ;
<MentalAtom>     = "true" | "bel" <LitConj> | "goal" <LitConj> ;
<Action>         = <UserDefAction> | <BuiltInAction> ;
<UserDefAction>  = <Atom> [ <Parameters> ] ;
<BuiltInAction>  = "insert(" <PosLitConj> ")"
                 | "delete(" <PosLitConj> ")"
                 | "adopt(" <PosLitConj> ")"
                 | "drop(" <PosLitConj> ")"
                 | "delete(" <Id> "," <PosLitConj> ")" ;
<Id>             = ? an arbitrary identifier ? ;
```

A.3. Jason

A.3.1. MAS-Files

```
<MAS>              = "MAS" <ID> "{" ;
                     [ <Infrastructure> ]
                     [ <Environment> ]
                     [ <ExecControl> ]
                     <Agents> "}"
<Infrastructure>   = "infrastructure:" <ID> ;
<Environment>      = "environment:" <ID> [ "at" <ID> ] ;
<ExecControl>      = "executionControl:" <ID> [ "at" <ID> ] ;
<Agents>           = "agents:" { <Agent> }+ ;
<Agent>            = <ASID>
                     [ <FileName> ]
                     [ <Options> ]
                     [ "AgentArchClass" <ID> ]
                     [ "BeliefBaseClass" <ID> ]
                     [ "AgentClass" <ID> ]
                     [ "#" <NUMBER> ]
```

```
                    [ "at" <ID> ]
                    ";" ;
<FileName>        = [ <Path> ] <Ident> ;
<Options>         = "[" <Option> { "," <Option> } "]" ;
<Option>          = "events=" ( "discard" | "requeue" | "retrieve" )
                  | "intBels=" ( "sameFocus" | "newFocus" )
                  | "nrcbp=" <Number>
                  | "verbose=" <Number>
                  | <Ident> "=" ( <Ident> | <String> | <Number> )
<String>          = ? an arbitrary string ? ;
<Ident>           = ? an arbitrary identifier ? ;
<Path>            = ? an path ? ;
<Number>          = ? a number ? ;
;
```

A.3.2. Agent-Files

```
<Agent>         = <InitBeliefs> <InitGoals> <Plans> ;
<InitBeliefs>   = <Beliefs> <Rules> ;
<Beliefs>       = { <Literal> "." } ;
<Rules>         = { <Literal> ":-" <LogExpr> "." } ;
<InitGoals>     = { "!" <Literal> "." } ;
<Plans>         = { <Plan> } ;
<Plan>          = [ "@" <AtomicFormula> ] <TriggeringEvent>
                  [ ":" <Context> ] [ "<-" <Body> ] "." ;
<TriggeringEvent>= ( "+" | "-" ) [ "!" | "?" ] <Literal> ;
<Literal>       = [ "~" ] <AtomicFormula> ;
<Context>       = <LogExpr> | "true" ;
<LogExpr>       = <SimpleLogExpr> | "not" <LogExpr>
                | <LogExpr> "&" <LogExpr>
                | <LogExpr> "|" <LogExpr>
                | "(" <LogExpr> ")" ;
<SimpleLogExpr> = ( <Literal> | <RelExpr> | <Var> ) ;
<Body>          = <BodyFormula> { ";" <BodyFormula> }
                | "true" ;
<BodyFormula>   = ( "!" | "!!" | "?" | "+" | "-" | "-+" ) <Literal>
                | <AtomicFormula>
                | <Var>
                | <RelExpr> ;
<AtomicFormula> = ( <Atom> | <Var> )
                  [ "(" <ListOfTerms> ")" ]
                  [ "[" <ListOfTerms> "]" ] ;
<ListOfTerms>   = <Term> { "," <Term> } ;
```

```
<Term>          = <Literal> | <List> | <ArithmExpr> | <Var> | <String> ;
<List>          = "["
                  [ <Term> ( "," <Term> ) [ "|" ( <List> | <Var> ) ] ]
                  "]" ;
<RelExpr>       = <RelTerm>
                  { ( "<" | "<=" | ">" | ">=" | "==" | "\==" | "=" | "=.." )
                  <RelTerm> }+ ;
<RelTerm>       = <Literal> | <ArithmExpr> ;
<ArithmExpr>    = <ArithmTerm>
                  { ( "+" | "-" | "*" | "**" | "/" | "div" | "mod" )
                  <ArithmTerm> }+ ;
<ArithmTerm>    = <Number> | <Var>
                | ( "-" <ArithmExpr> )
                | ( "(" <ArithmExpr> ")" ) ;
<Var>           = ? a string starting with an upper-case letter ? ;
<Atom>          = ? a logic-programming like atom ? ;
<String>        = ? an arbitrary string ? ;
<Number>        = ? a number ? ;
```

B. Environment Programming Classes

B.1. 2APL

```
public abstract class Environment {

    private HashMap<String,APLAgent> agents =
        new HashMap<String,APLAgent>();
    public final void addAgent(APLAgent agent) { ... }
    public final void removeAgent(APLAgent agent) { ... }
    protected abstract void addAgent(String name);
    protected abstract void removeAgent(String name);
    protected final void throwEvent(APLFunction e, String... receivers)
        { ... }
    public final String getName() { ... }
    public void takeDown() { ... }

}
```

B.2. Goal

```
public interface Environment {

    public boolean executeAction(String pAgent, String pAct)
        throws Exception;
    public ArrayList<Percept> sendPercepts(String pAgentName)
        throws Exception;
    public boolean availableForInput();
    public void close();
    public void reset() throws Exception;

}
```

B.3. Jason

```
public class Environment {
```

```
    private static Logger logger =
        Logger.getLogger(Environment.class.getName());
    private List<Literal> percepts =
        Collections.synchronizedList(new ArrayList<Literal>());
    private Map<String,List<Literal>>  agPercepts =
        new ConcurrentHashMap<String, List<Literal>>();
    private boolean isRunning = true;
    private EnvironmentInfraTier environmentInfraTier = null;
    private Set<String> uptodateAgs =
        Collections.synchronizedSet(new HashSet<String>());
    protected ExecutorService executor;

    public Environment(int n) { ... }
    public Environment() { ... }
    public void init(String[] args) { ... }
    public void stop() { ... }
    public boolean isRunning() { ... }
    public void setEnvironmentInfraTier(EnvironmentInfraTier je) { ... }
    public EnvironmentInfraTier getEnvironmentInfraTier() { ... }
    public Logger getLoger() { ... }
    public void informAgsEnvironmentChanged(Collection<String> agents)
        { ... }
    public void informAgsEnvironmentChanged() { ... }
    public List<Literal> getPercepts(String agName) { ... }
    public void addPercept(Literal per) { ... }
    public boolean removePercept(Literal per) { ... }
    public int removePerceptsByUnif(Literal per) { ... }
    public void clearPercepts() { ... }
    public boolean containsPercept(Literal per) { ... }
    public void addPercept(String agName, Literal per) { ... }
    public boolean removePercept(String agName, Literal per) { ... }
    public int removePerceptsByUnif(String agName, Literal per) { ... }
    public boolean containsPercept(String agName, Literal per) { ... }
    public void clearPercepts(String agName) { ... }
    public void scheduleAction(final String agName,
        final Structure action, final Object infraData) { ... }
    public boolean executeAction(String agName, Structure act) { ... }

}
```

C. EIS and APL Extensions

C.1. The Environment Interface Standard

```
package eis;

import java.util.Collection;
import java.util.Map;

import eis.exceptions.ActException;
import eis.exceptions.AgentException;
import eis.exceptions.EntityException;
import eis.exceptions.ManagementException;
import eis.exceptions.NoEnvironmentException;
import eis.exceptions.PerceiveException;
import eis.exceptions.RelationException;
import eis.iilang.Action;
import eis.iilang.EnvironmentState;
import eis.iilang.Parameter;
import eis.iilang.Percept;

public interface EnvironmentInterfaceStandard {

        void attachEnvironmentListener(EnvironmentListener listener);
        void detachEnvironmentListener(EnvironmentListener listener);
        void attachAgentListener(String agent, AgentListener listener);
        void detachAgentListener(String agent, AgentListener listener);

        void registerAgent(String agent) throws AgentException;
        void unregisterAgent(String agent) throws AgentException;
        Collection<String> getAgents();
        Collection<String> getEntities();
        void associateEntity(String agent, String entity)
            throws RelationException;
        void freeEntity(String entity) throws RelationException;
        void freeAgent(String agent) throws RelationException;
        void freePair(String agent, String entity)
            throws RelationException;
        Collection<String> getAssociatedEntities(String agent)
```

```
            throws AgentException;
        Collection<String> getAssociatedAgents(String entity)
            throws EntityException;
        Collection<String> getFreeEntities();
        String getType(String entity) throws EntityException;

        Map<String,Percept> performAction(String agent, Action action,
                        String... entities) throws ActException;
        Map<String,Collection<Percept>> getAllPercepts(String agent,
            String... entities) throws PerceiveException,
            NoEnvironmentException;
        boolean isStateTransitionValid(EnvironmentState oldState,
            EnvironmentState newState);

        void init(Map<String,Parameter> parameters)
            throws ManagementException;
        void start() throws ManagementException;
        void pause() throws ManagementException;
        void kill() throws ManagementException;
        EnvironmentState getState();
        boolean isInitSupported();
        boolean isStartSupported();
        boolean isPauseSupported();
        boolean isKillSupported();

        String requiredVersion();

        String queryProperty(String property);
        String queryEntityProperty(String entity, String property);

}
```

D. APLTK

D.1. Tool Interface

```
public interface Tool {

    public void init(Element parameters) throws ToolException;
    public void addInterpreters(Collection<Interpreter> interpreters);
    public void addEnvironments(Collection<EnvironmentInterfaceStandard>
        environments);
    public void processStepResults(Collection<StepResult> stepResults);
    public void release();
    public void setBasePath(String basePath);
    public void setCore(Core core);

    public void handleStarted();
    public void handlePaused();
    public void handleFinished();
    public void handleSignal(String sender, String signal);

    public ToolWindow getWindow();
    public String getName();

}
```

D.2. Interpreter Interface

```
public interface Interpreter {

    public void init(Element parameters) throws InterpreterException;
    public void addEnvironment(EnvironmentInterfaceStandard env);
    public StepResult step();
    public void release();

    public Collection<String> getAgents();
    public QueryCapabilities getQueryFlags();
    public Collection<Belief> getBeliefBase(String agent);
    public Collection<Goal> getGoalBase(String agent);
```

```
    public Collection<Plan> getPlanBase(String agent);
    public Collection<Event> getEventBase(String agent);
    public Collection<Percept> getPerceptBase(String agent);
    public Collection<Message> getMessageBox(String agent);
    public Collection<Coalition> getCoalitions();

    public String getName();
    public void setBasePath(String basePath);

}
```

Bibliography

[1] The Foundation for Intelligent Physical Agents. `http://www.fipa.org/`.

[2] S. Albayrak and D. Wieczorek. JIAC - a toolkit for telecommunication applications. In S. Albayrak, editor, *IATA*, volume 1699 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1999.

[3] R. C. Arkin. *Behavior-Based Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, May 1998.

[4] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003. ISBN 0521818028.

[5] T. Behrens. Agent-oriented control in real-time computer games. In L. Braubach, J.-P. Briot, and J. Thangarajah, editors, *Proceedings of 7th international Workshop on Programming Multi-Agent Systems, ProMAS 2009*, volume 5919 of *LNAI*, Heidelberg, Germany, 2010. Springer.

[6] T. Behrens, R. Bordini, L. Braubach, M. Dastani, J. Dix, K. Hindriks, J. Hübner, and A. Pokahr. An interface for agent-environment interaction. In *Proceedings of 8th international Workshop on Programming Multi-Agent Systems, ProMAS 2010*, volume 6599 of *LNCS*, pages 170–185. Springer, 2011.

[7] T. Behrens, M. Dastani, J. Dix, M. Köster, and P. Novák, editors. *Special Issue about Multi-Agent-Contest*, volume 59 of *Annals of Mathematics and Artificial Intelligence*. Springer, Netherlands, 2010.

[8] T. Behrens, M. Dastani, J. Dix, and P. Novák. Agent contest competition - 4th edition. In *Proceedings of Sixth international Workshop on Programming Multi-Agent Systems, ProMAS'08*, volume 5442 of *LNAI*. Springer, 2008.

[9] T. Behrens, M. Dastani, J. Dix, and P. Novák. Agent contest competition: 4th edition. In K. V. Hindriks, A. Pokahr, and S. Sardiña, editors, *Programming Multi-Agent Systems, 6th International Workshop (ProMAS 2008)*, volume 5442 of *Lecture Notes in Computer Science*, pages 211–222. Springer, 2009.

[10] T. Behrens, J. Dix, and K. V. Hindriks. The environment interface standard for agent-oriented programming platform integration guide and interface implementation guide. Technical Report IfI-09-10, Clausthal University of Technology, 2009.

[11] T. Behrens, J. Dix, and K. V. Hindriks. Towards an environment interface standard for agent-oriented programming. Technical Report IfI-09-09, Clausthal University of Technology, 2009.

[12] T. Behrens, K. Hindriks, and J. Dix. Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, 61:3–38, 2011.

[13] T. Behrens, K. Hindriks, J. Hübner, and M. Dastani. Putting APL platforms to the test: Agent similarity and execution performance. Technical Report IfI-10-09, Clausthal University of Technology, 2010.

[14] T. Behrens, R. Schärfig, and T. Winkler. Multi-agent navigation using path-based vector fields. In L. Braubach, W. van der Hoek, P. Petta, and A. Pokahr, editors, *Multiagent System Technologies (MATES)*, volume 5774 of *Lecture Notes in Computer Science*, pages 4–15. Springer, 2009.

[15] F. Bellifemine and G. Rimassa. Developing multi-agent systems with a FIPA-compliant agent framework. *Softw. Pract. Exper.*, 31:103–128, February 2001.

[16] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[17] R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Programming Multi Agent Systems: Languages, Platforms and Applications*, volume 15 of *Multi-agent Systems, Artificial Societies and Simulated Organizations*. Springer, Berlin, 2005.

[18] R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors. *Multi-Agent Tools: Languages, Platforms and Applications*. Springer, Berlin, 2009.

[19] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[20] M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

[21] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.

[22] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. JACK intelligent agents - components for intelligent agents in java, 1999.

[23] M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[24] M. Dastani, J. Dix, and P. Novák. The first contest on multi-agent systems based on computational logic. In F. Toni and P. Torroni, editors, *Computational Logic*

in Multi-Agent Systems, 6th International Workshop, CLIMA VI, volume 3900 of *Lecture Notes in Computer Science*, pages 373–384. Springer, 2005.

[25] M. Dastani, J. Dix, and P. Novák. The second contest on multi-agent systems based on computational logic. In K. Inoue, K. Satoh, and F. Toni, editors, *Computational Logic in Multi-Agent Systems, 7th International Workshop, CLIMA VII*, volume 4371 of *Lecture Notes on Computer Science*, pages 266–283. Springer, 2006.

[26] M. Dastani, J. Dix, and P. Novák. Agent contest competition - 3rd edition. In M. Dastani, A. Ricci, A. El Fallah Seghrouchni, and M. Winikoff, editors, *Proceedings of ProMAS '07, Revised Selected and Invited Papers*, number 4908 in Lecture Notes in Artificial Intelligence, Honululu, US, 2008. Springer.

[27] M. Dastani et al. *2APL Manual*. `http://www.cs.uu.nl/2apl/`.

[28] M. Dastani, M. B. van Riemsdijk, F. Dignum, M. Birna, R. F. Dignum, and J.-J. C. Meyer. A programming language for cognitive agents goal directed 3APL. pages 111–130. Springer, 2003.

[29] F. S. de Boer, K. V. Hindriks, W. van der Hoek, and J.-J. C. Meyer. Agent programming with declarative goals. *CoRR*, cs.AI/0207008, 2002.

[30] J. Dix and Y. Zhang. IMPACT: A multi-agent framework with declarative semantics. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Programming Multi Agent Systems: Languages, Platforms and Applications*, volume 15, pages 69–94. Springer, Berlin, 2005.

[31] J. Ferber. *Multi-Agent Systems: An Introduction to Artificial Intelligence*. Addison-Wesley, 1999. ISBN 0-201-36048-9.

[32] J. Hagelbäck and S. J. Johansson. Dealing with fog of war in a real time strategy game environment. In *Proceedings of 2008 IEEE Symposium on Computational Intelligence and Games (CIG)*, 2008.

[33] J. Hagelbäck and S. J. Johansson. Using multi-agent potential fields in real-time strategy games. In *AAMAS (2)*, pages 631–638, 2008.

[34] K. Hindriks. The GOAL programming guide, March 2009.

[35] K. V. Hindriks. Programming rational agents in GOAL. In A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini, editors, *Multi-Agent Programming:*, pages 119–157. Springer US, 2009.

[36] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Agent programming in 3APL. *AAMAS Journal*, 2:357–401, 1999.

[37] K. V. Hindriks, M. B. van Riemsdijk, T. Behrens, R. Korstanje, N. Kraaijenbrink, W. Pasman, and L. de Rijk. Unreal GOAL agents. In *AGS 2010*, 2010.

[38] J. F. Hübner and J. S. Sichman. Saci: Uma ferramenta para implementação e monitaração da comunicação entre agentes. In M. C. Monard and J. S. Sichman, editors, *IBERAMIA-SBIA 2000 Open Discussion Track*, pages 47–56. ICMC/USP, 2000.

[39] N. R. Jennings. Building complex software systems: The case for an agent-based approach. *Communications of the ACM, Forthcoming*, 44:12–23, 2000.

[40] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. volume 2, pages 500–505, 1985.

[41] Y. Koren, S. Member, and J. Borenstein. Potential field methods and their inherent limitations for mobile robot navigation. In *In Proc. IEEE Int. Conf. Robotics and Automation*, pages 1398–1404, 1991.

[42] B. Krogh. A generalized potential field approach to obstacle avoidance control. 1984.

[43] B. Lars, P. Alexander, and L. Winfried. Jadex: A BDI-agent system combining middleware and reasoning. In V. R. Unland, M. Klusch, and M. Calisti, editors, *Software agent-based applications, platforms and development kits*, 2005.

[44] M. Mamei and F. Zambonelli. Field-based motion coordination in Quake 3 Arena. In *AAMAS*, pages 1532–1533. IEEE Computer Society, 2004.

[45] M. Massari, G. Giardini, and F. Bernelli-Zazzera. Autonomous navigation system for planetary exploration rover based on artificial potential fields. In *Proceedings of Dynamics and Control of Systems and Structures in Space (DCSSS) 6th Conference*, 2005.

[46] P. Novák. *Behavioural State Machines - Agent Programming and Engineering*. PhD thesis, Institut für Informatik, TU Clausthal, 2009.

[47] G. M. P. O Hare. Agent Factory: An Environment for the Fabrication of Distributed Artificial Systems. *O Hare, Gregory M. P. and Jennings, N. R. (Eds.), Foundations of Distributed Artificial Intelligence, Sixth Generation Computer Series, Wiley Interscience Pubs*, pages 449–484, 1996.

[48] L. of Pisa. *Liber Abaci*. 1202.

[49] W. Pasman. GOAL IDE user manual, September 2009. `http://mmi.tudelft.nl/~koen/goal.php`.

[50] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. W. Perram, editors, *MAAMAW*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996.

[51] A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95*, pages 312–319, 1995.

[52] A. Ricci, M. Piunti, D. L. Acay, R. H. Bordini, J. F. Hübner, and M. Dastani. Integrating heterogeneous agent programming platforms within artifact-based environments. In L. Padgham, D. C. Parkes, J. P. Müller, and S. Parsons, editors, *AAMAS (1)*, pages 225–232. IFAAMAS, 2008.

[53] A. Ricci, M. Piunti, L. D. Acay, R. H. Bordini, J. F. Hübner, and M. Dastani. Integrating heterogeneous agent programming platforms within artifact-based environments. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 225–232, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[54] A. Ricci, M. Viroli, and A. Omicini. Environment-based coordination through coordination artifacts. In D. Weyns, H. V. D. Parunak, and F. Michel, editors, *E4MAS*, volume 3374 of *Lecture Notes in Computer Science*, pages 190–214. Springer, 2004.

[55] S. J. Russell and Norvig. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall, 2003.

[56] S. Sardiña, G. D. Giacomo, Y. Lespérance, and H. J. Levesque. On the semantics of deliberation in indigolog - from theory to implementation. *Ann. Math. Artif. Intell.*, 41(2-4):259–299, 2004.

[57] Y. Shoham. AGENT0: a simple agent language and its interpreter. In *Proceedings of the ninth National conference on Artificial intelligence - Volume 2*, AAAI'91, pages 704–709. AAAI Press, 1991.

[58] M. Sierhuis. *Modeling and Simulating Work Practice; Brahms: A multiagent modeling and simulation language for work system analysis and design.* Phd thesis, University of Amsterdam, SIKS Dissertation Series No. 2001-10, 2001.

[59] L. Sterling and E. Shapiro. *The art of Prolog: advanced programming techniques.* MIT Press, Cambridge, MA, USA, 1986.

[60] L. Suchman. Representations of work. *Communications of the ACM/Special Issue 38(9)*, 1995.

[61] A. Suna and A. E. Fallah-Seghrouchni. A mobile agents platform: Architecture, mobility and security elements. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *PROMAS*, volume 3346 of *Lecture Notes in Computer Science*, pages 126–146. Springer, 2004.

[62] C. Ullenboom. *Java ist auch eine Insel.* Galileo Computing, Bonn, 6th edition, 2007.

[63] B. van Riemsdijk, W. van der Hoek, and J.-J. C. Meyer. Agent programming in dribble: from beliefs to goals using plans. In *AAMAS*, pages 393–400. ACM, 2003.

[64] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2002.

[65] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

List of Figures

List of Tables

Short Curriculum Vitæ

Personal Details

Date and place of birth:	June 16th 1981, Osterode am Harz
Nationality:	German

Education

2000	Abitur
2000 - 2001	Military Service
2001 - 2006	Studies of Computer Science Clausthal University of Technology Diploma thesis: *Statische Analyse von Spec-Sharp-Programmen*
2006 - present	PhD Studies Department of Computer Science Clausthal University of Technology

Professional Experiences

2006 - present	Scientific Employee Department of Computer Science Clausthal University of Technology